Question-learn-test-feedback pattern to test emerging software construction paradigms

Benoit Baudry

Année 2010

Habilitation à diriger des recherches / UNIVERSITÉ DE RENNES 1

sous le sceau de l'Université Européenne de Bretagne

Mention : Informatique

Présentée par

Benoit Baudry

préparée à l'unité de recherche UMR 6074 IRISA

Institut de Recherche en Informatique et Systèmes Aléatoires

Composante Universitaire : IFSIC

Habilitation soutenue à Rennes
le 10 décembre 2010

devant le jury composé de :

Antonia BERTOLINO
Directrice de Recherhe à CNR, Italie / rapporteur

Yves LEDRU
Professeur à l'Université Joseph Fourier / rapporteur

Jeff OFFUTT
Professeur à Georges Mason University, USA / rapporteur

Bernhard RUMPE
Professeur à Aachen Universität, Allemagne / rapporteur

Pierre COINTE
Professeur à l'École des Mines de Nantes / examinateur

Thomas JENSEN
Directeur de Recherche à l'INRIA / examinateur

Jean-Marc JÉZÉQUEL
Professeur à l'Université de Rennes 1 / examinateur

Question-learn-test-feedback pattern to test emerging software construction paradigms

Contents

Chapter 1

Introduction

> *Myers in 1979 presents software testing as: "Testing is the process of executing a program with the intent of finding errors" [109]*
>
> *Bertolino introduces a much broader definition at ICSE'07: "Software testing is a broad term encompassing a wide spectrum of different activities, from the testing of a small piece of code by the developer (unit testing), to the customer validation of a large information system (acceptance testing), to the monitoring at run-time of a network-centric service-oriented application." [29]*
>
> *Amman and Offutt in 2008 present "... software testing as a practical engineering discipline, essential to producing high-quality software." They also mention "the complex and confusing landscape of test coverage criteria". [5]*

These definitions of software testing, given 30 years apart, illustrate how much the scope and diversity of software testing techniques have expanded. This expansion has accompanied the increasing variety of software construction languages, methods and tools that appeared in order to face the growing heterogeneity and complexity of demands regarding software systems. Today everyone expects software systems to deal with financial networks, accurate medical images, embedded calculators, smart phones and alarm clocks. This translates in heterogeneous requirements that encompass concerns as varied as performance (more data, more services, more speed and constant reliability), security, usability and adaptability. In addition, software systems have to run on heterogeneous hardware platforms, interact through a variety of devices (who can imagine that there once was a time when it was impossible to plug a smart phone into a car to listen to music?) and deliver continuous on-demand services.

The metamorphosis in the scope and range of software systems has forced engineers and researchers to continuously evolve software construction paradigms. In the past decades the fundamental principles of abstraction, modularity and separation of concerns [68] have been incarnated in a variety of programming languages, architecture styles and software development methods. For example object-oriented programming languages and design methods have evolved towards aspect-oriented and model-driven

software development. This evolution has to continue as emphasized by Wirsing et al. in their analysis of challenges for software intensive systems [146] or as proposed by Carlo Ghezzi in his SMSCom project [67].

Testing techniques have to consider the specificities of the new paradigms and methods in order to leverage the models they introduce for software systems. For example, considering that a program can be decomposed into classes that declare a set of operations and that encapsulate a set of data, has an impact on the way test cases can be organized, on the data they can observe and the operations they can execute. JUnit [26] is a nice illustration of a testing technique that has appeared, expanded and grown extremely popular by taking into account the specificities of the object-oriented construction paradigm. It clearly leverages the class as the unit for testing. By forcing test cases to observe a class from the client's point of view it respects the object-oriented philosophy of encapsulation. It is also interesting to notice that the JUnit framework's implementation is a remarkable application of object-oriented best practices.

When a construction paradigm is mature enough, testing can focus on its initial error-detection purpose. For example, in the context of embedded programs, there exist standard methods that impose strict sequences of requirements definition, design and development. Thus, a tester knows exactly what is available for testing and can leverage this to develop advanced techniques that target efficient error detection and localization. However, when paradigms are not completely stable some elements might be missing or difficult to understand for testing. For example, it is still not clear how the behavior of an aspect-oriented program should be specified or how system requirements can be the most appropriately separated into a set of architectural elements. These uncertainties in evolving paradigms have a major impact on the definition of an oracle, test adequacy criteria, program analysis and on the testing activity as a whole.

In this habilitation I defend the following thesis

> **Software testing research for evolving construction paradigms has to extend beyond error detection and must spend some effort for understanding the scope of these paradigms.**

This means it is necessary to ask questions about the assumptions the paradigms make and the elements they introduce for software construction. In some cases, answers to these questions are straightforward and we develop effective testing techniques to detect the specific errors introduced by the paradigm. In other cases, it is necessary to perform more in-depth investigations of the paradigm itself in order to understand how testing techniques should be integrated. These empirical studies that aim at understanding testability of evolving paradigms is a special case of experiments in software engineering as defined by Basili et al. [9]. These studies result in two types of output: add new construction mechanisms to the paradigm; develop testing techniques that specifically target error detection for the paradigm. In summary, software testing research for evolving construction paradigms has to investigate two levels: testing the paradigm itself (to understand and develop it); testing the software systems built according to this paradigm.

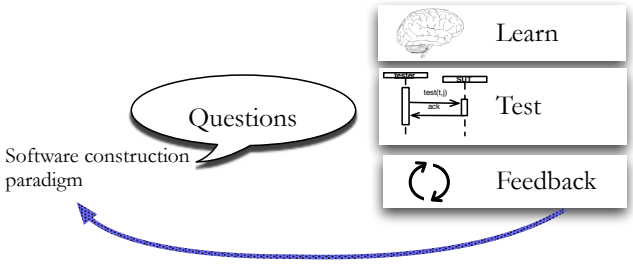

Figure 1.1: The QLTF pattern to investigate testing in evolving software construction paradigms

1.1 Following a question-learn-test-feedback pattern

This habilitation thesis summarizes a series of investigations about software testing in different software construction paradigms. In particular, we show how these investigations followed a common pattern in order to understand how software systems could be analyzed and tested in these paradigms. The QLTF pattern is displayed in figure 1.1 and captures this common path .

- Questions. When developing testing techniques in a new paradigm, we first have to understand what this paradigm imposes on the software, the construction process and the verification phases. The initial step for investigating a paradigm is thus a series of questions. These questions can be about the models this paradigm produces in order to analyze the system or what types of errors can occur in this context. We also had questions about the constraints the paradigm imposes on software construction and how these constraints can prevent errors or do hamper the testability of the system. Answers to these questions can lead to develop the following three points.

- Learn. This part of the pattern captures contributions about the understanding of the paradigm. When answering the questions about the paradigms, we performed several types of actions such as literature survey, empirical analyses or testability analysis. These actions led to different types of results such as the systematic identification of barriers for testing or a quantitative evaluation of a paradigm [108]. The 'Learn' part of the pattern synthesizes these outputs.

- Test. Our initial intent for investigating software construction paradigms is to develop testing techniques that can effectively detect error in systems developed inside these paradigms. This part of the pattern presents the contributions we have had in terms of software testing, leveraging the specificities of each paradigm. We present new techniques for test data generation, adequacy criteria or oracle specification.

- Feedback. It happens that when studying the impact a paradigm has on software development we discover gaps. Sometimes these gaps are limitations to the development of testing techniques. If this occurs, we have to address the gaps before studying the way systems can be tested. Our contributions to the paradigm in order to address these gaps fall in the 'feedback' part of the pattern.

1.1.1 A brief discussion about methods

The organization of this work along a systematic pattern for investigation emphasizes a methodological contribution rather than a specific technical proposal for testing. This choice highlights the principle that drives our work: when one aims at defining testing techniques for a new paradigm, it is necessary to avoid any assumption about the relevance of this paradigm. On the contrary, understanding to what extent a paradigm can improve the construction of correct software systems and in which cases it is not well suited should be a part of the investigation for testing techniques. This understanding allows setting the precise context in which testing is performed. Knowledge about this context reinforces the utility of the new testing techniques by fitting them into concrete cases where the paradigm is used. Since this document defends a methodological approach for software testing research in the context of new construction paradigms, I briefly discuss our work with respect to processes for scientific investigation.

We can distinguish two levels in the QLTF pattern. *Question* and *Learn* operate at the meta level, in the sense that their subject of study is the paradigm itself. We question, formulate hypothesis and analyze the construction paradigm. This can be related to scientific processes as found in natural sciences where the paradigm would be a theory and *Question* and *Learn* aim at understanding, consolidating or refuting the theory. As a matter of fact, the different incarnations of *Learn*ing can be related to the falsifiability test promoted by Karl Popper in order to establish a theory as scientific [118]. The goal of the QLTF pattern was to develop testing techniques tightly integrated in new software construction paradigms. In order to achieve this goal we needed to understand how testing could be performed in this paradigm and identify what attributes could help or hinder testability. The identification of these attributes is part of the process proposed by Popper for a scientific investigation. On the other hand, T and F operate at the concrete level in the sense that their subject of study is the software system that is constructed following a given paradigm.

Going a bit further into the epistemic perspective it must also be noted that, even if a part of the pattern can be related to a well defined scientific approach on software construction paradigms, the emergence of the pattern itself is closer to Feyerabend's anarchist perspective on scientific discovery [56]. Indeed, the pattern provides a synthetic template in which we can organize our work and promote an idea: software testing has to be integrated in evolving software construction paradigms. However, this pattern emerged from our investigations but it did not precede our research on software testing. We had an intuition about the necessity to go through these steps and investigate in these different directions in order to propose testing techniques fitted to new paradigms, but

we discovered the pattern only later. Put in Feyerabend's words, action (the 'irrational', loosely related series of work we performed) preceded the idea (the QLTF pattern that organizes these investigations).

1.2 Software construction paradigms under study

Figures 1.2 present the instantiation of the QLTF pattern in three paradigms, discussed in more details below.

This work starts with the investigation of the object-oriented paradigm for analyzing, designing and implementing software systems. The emergence of object-oriented languages has led to numerous work, concerns, studies, skepticism and new methodologies from the testing research community. For example, one can look at Binder's book [32] to understand the huge impact that this paradigm had on the testing research and practice. When our work started in the early 2000's, programming languages were pretty stable, but numerous development methods, reusable design solutions were emerging and the paradigm was still under construction. For example, here was still some room to understand the role of design patterns and Design by Contract for testing.

The growing interest in object-oriented design and modelling with the UML led more attention to model-driven engineering [123] for software systems and one of its standardized incarnation, MDA [111]. Approximately at the same time, AspectJ [78] appeared and initiated important explorations towards advanced separation of concerns and aspect-oriented software development. It is interesting to notice that these two evolutions clearly complement each other. On one hand, model-driven engineering (MDE) emphasizes the use of models to reason on a software system at different levels of abstraction. In this context, models are abstract representation of the system for a specific purpose. Thus, there are different models to reason on different parts of the system. On the other hand, aspect-oriented software development (AOSD) emphasizes the need to separate concerns and investigate advanced mechanisms to improve modularity in large software systems. These paradigms complement each other in at least two ways. First, the different purposes for which one would like an abstract model in MDE can often be related to a concern AOSD would like to identify and separate from the rest of the system [75]. Second, Ghezzi et al. advocate abstraction, modularity and separation of concerns as major principles for software engineering [68]. It appears that the association of MDE and AOSD aims at deploying these principles.

The conceptual and technical foundations of MDE and AOSD are well established and accepted as solutions to cope with the complexity of software systems. However, there remain a large number of challenges [65]. In this work we investigate testing in different two specific areas of MDE and AOSD: aspect-oriented programming, model transformations.

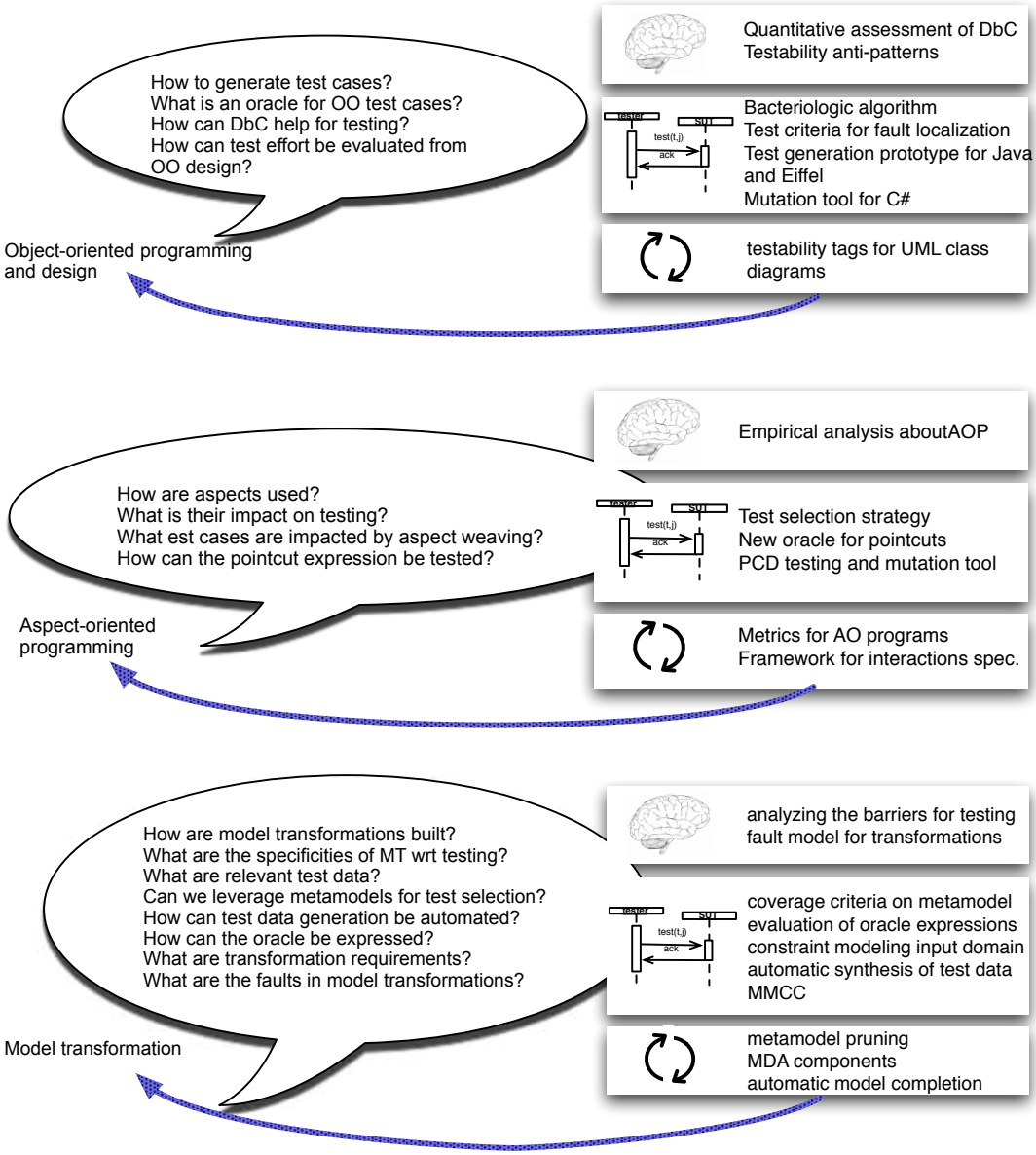

Figure 1.2: Instantiating the QLTF pattern in three construction paradigms

1.3 Supervising research

The work presented here and the emergence of the QLTF pattern result from collaborations I have had with researcher colleagues as well as students I supervised during their Masters and PhD thesis. Table 1.1 gives the list of PhD I co-supervised: it provides the amount of co-supervision work I took care of, the period, the defense date and the topic.

Name	Rate (%)	Period	Defense	Topic
Franck Fleurey	50	2003 – 06	09/10/06	Model transformation
Jean-Marie Mottu	80	2005 – 08	26/11/08	Model transformation
Erwan Brottier	70	2005 – 08	10/12/09	Model composition
Romain Delamare	80	2006 – 09	02/12/09	AOP
Sagar Sen	80	2007 – 10	22/06/10	Model transformation
Tejedinne Mouelhi	30	2007 – 10	22/09/10	Security
Freddy Muñoz	80	2007 – 10	29/09/10	Adaptive systems and AOP
Juan Cadavid	80	2009 – 12		Metamodeling
Nicolas Sannier	50	2010 – 13		Requirements engineering
Aymeric Hervieu	30	2010 – 13		Software product lines
Olivier Bendavid	80	2010 – 13		Security

Table 1.1: PhD co-supervision from 2004 to 2010

1.4 Organization of the thesis

In the remaining of this document I present three incarnations of the QLTF pattern in object-oriented programming and design (chapter 2), aspect-oriented programming (chapter 3) and model transformation (chapter 4). In each chapter I summarize the questions we investigated in the paradigm and the contributions in the *Learn*, *Test* and *Feedback* facets of the pattern. I conclude each chapter with a brief section that provides a retrospective interpretation of these work. Chapter 5 concludes and opens a set of perspectives about V&V and MDE for heterogeneous software-intensive systems.

Chapter 2

Object-oriented Design

Figure 2.1: Investigating object-oriented software development

In the late 90's, Beck claimed that "programmers love writing tests" [24] in the object-oriented paradigm. One reason for this is that they can incrementally build confidence in their code when it passes their tests. If one can qualify the relevance of test cases, the level of confidence one has in a program can be linked to the quality of its test cases. *Mutation analysis* [47, 79, 95], is a test qualification technique based on the observation that the quality of the test cases is related to the proportion of faulty programs (also called *mutants*) it detects. Faulty programs are generated by systematic fault injection in the original implementation. By measuring the quality of test cases (the revealing power of the test cases [144]), we seek to build trust in a component passing those test cases. The test cases that are generally provided by the tester easily cover 50-70% of the mutants, but improving this score up to 90-100% is a time-consuming and a very expensive task. The first question we investigated in the OO paradigm is: how can we automatically improve the quality of test cases for object-oriented program?

We investigated the structure of object-oriented programs and associated test cases, and realized that genetic algorithms were not well suited for this optimization problem

[10, 13]. This observation led to the definition of a variant of the genetic algorithm called bacteriologic algorithm that proved to be effective at automatically improving the fault detection power of an initial test set [15]. This algorithm, that will detailed in section 2.2.1, augments the initial set with additional test sequences and test data, but there remains one issue to have complete test cases: how can we build an oracle for these new test sequences? Here again, we analyzed a specific objet-oriented design method called Design by Contract ©(DbC). This led us to study how DbC can help testing object-oriented programs. In particular, we wondered how contracts could help in detecting and locating errors when executing a test suite [17, 86] (section 2.1.2). The ability of contracts to detect errors makes them excellent candidates for the oracle of test cases for object-oriented programs. Their ability to locate errors is an added value, which can greatly help when debugging the program after detecting the presence of an error.

We also study the impact of design patterns as another major mechanism for designing large object-oriented systems. Design patterns solve recurrent design problems through an extensive use of delegation and inheritance. This results in a code that is much smaller and simple, but in a design that locally increases the coupling between some modules. From our tester's point of view this seems to be counter-productive as it decreases the testability and thus increases the cost and effort for testing. The third question we investigate for OO design is: how can the testing effort be estimated on early design and how can it be mitigated [20]? This is detailed in section section 2.1.1.

Answers to the previous questions have led to original contributions for testing object-oriented programs. They also generated new knowledge about object-oriented programming and design methods, which allowed us to propose original contributions to the object-oriented paradigms. This investigation thus led through the whole spectrum of the triptych pattern, as shown in figure 2.1. These contributions results mostly from my PhD thesis and Franck Fleurey's Master thesis. They are summarized in the following sections.

2.1 What we learned about the OO paradigm

In our investigation of the object-oriented paradigm and how it can be leveraged for testing, we had to learn about the different methods and techniques that emerge when building software systems following an object-oriented style. In particular, we focused an important of our investigation on design patterns and Design by Contract ©. As a result of these studies we were able to quantify the effect of these approaches on the quality of an object-oriented design. We measure testability as a quality factor for design patterns [18], and propose metrics of vigilance and diagnosability for Design by Contract ©[17, 86].

2.1.1 Testability anti-patterns in UML class diagrams

Any technique that improves a software design at an early stage can have highly beneficial impact on the final testing cost and efficiency. We have studied the testability of

UML class diagrams, looking for parts of the architecture where complex interactions may appear and lead to difficulties for testing. Testability can be informally defined as the easiness to test a piece of software, is a strongly desired feature of software.

In this work we identify two main anti-patterns that have a negative impact on class diagram's testability: *class interactions* or *self-usage interactions*. The global test cost estimate is related to the number of detected testability anti-patterns. We formally define the anti-patterns, the test cost and the global testability of a class diagram using a graph-based formalism [18]. Intuitively the *class interaction* anti-pattern occurs when there can be at least two different dependency paths from a class A to another class B. The *self-usage interaction* anti-pattern occurs when there exists at least one dependency path from a class A to itself. Since the interactions occur on the class diagram, but dynamic testing is concerned about interactions between running objects, it is necessary to distinguish between class and object levels for both anti-patterns. It is then possible to define the following test criterion

Definition 1. ***Test criterion***. *For each class interaction, either a test case is produced that exhibits a corresponding object interaction, either a report is produced that shows this interaction is not feasible.*

All these definitions allowed us to formally define the complexity of testability anti-patterns, the number of occurrences in a class diagram and thus an upper bound of the testing cost implied by the model.

A second lesson learnt from this work about object-oriented models was a more precise understanding of the impact of design patterns on object-oriented models. The UML standard allowed the development of systematic methodologies [49, 63] for object-oriented modelling that offer a decomposition approach for the architecture or guidelines to deal with evolution. These methodologies help design object-oriented software as a sequence of refinements, from initial analysis to the implementation. In particular, design patterns [66] may serve as a basis for such a refinement. Design patterns then correspond to subsets in the class diagram, and can be considered as intermediate structures between the overall architecture and the single class. This system decomposition provides an interesting solution, at a local level, for problems that are too complex at the global level.

In order to integrate the testability improvement in the design process, we consider that each refinement of a class diagram (due to the application of one or several design patterns) must lead to another testable class diagram or to a decision of introducing a testability weakness. Considering that design patterns are a usual practice for refining class diagrams, we build a catalogue that establishes the relationship between a chosen design pattern, the parameters that impact the testability and the corresponding value of testability as shown 3. More details can be found in a paper published at METRICS'03 [22].

Design Pattern	**Number of participants**	**# paths in a class interaction**	**# cycles in a self usage interaction**
Abstract Factory	1 client 1 abstract factory class n concrete factory m abstract products p concrete products	p≤ X ≤n*p	no
Decorator	1 interface 1 component class 1 abstract decorator class n concrete decorator classes	nor	1≤ X ≤n
Visitor	n visited elements (ConcreteElement) p visitors (ConcreteVisitor)	no	n*p

Table 2.1: Excerpt of a testability catalog for design patterns

2.1.2 Understanding the impact of contracts on vigilance and diagnosability

Design by Contract ©is a lightweight technique for embedding elements of formal specification (such as invariants, pre- and post-conditions) into an object-oriented design. When contracts are made executable, they can play the role of embedded, on-line oracles. Executable contracts allow components to be responsive to erroneous states, and thus may help in detecting and locating faults. In this work, we define vigilance as the degree to which a program is able to detect an erroneous state at runtime. Diagnosability represents the effort needed to locate a fault once it has been detected. In order to estimate the benefit of using Design by Contract ©, we formalize both notions of Vigilance and Diagnosability as software quality measures. The main steps of measure elaboration are given, from informal definitions of the factors to be measured to the mathematical model of the measures [87, 86]. As is the standard in this domain, the parameters are then fixed through actual measures, based on a mutation analysis in our case. Several measures are presented that reveal and estimate the contribution of contracts to the overall quality of a system in terms of vigilance and diagnosability.

The major results from this study are experimental demonstrations of the benefit of using contracts in order to improve software quality. Figure 2.2 summarizes the impact of contracts on the vigilance of three programs. The global vigilance corresponds to the ratio of errors that can be internally detected by the programs, whereas the isolated vigilance corresponds to the quality of the local contracts of each class. This chart indicates that even the introduction of poor contracts (e.g., that can detect 20% of the errors) on all classes can greatly improve the global vigilance of the program (if local contracts

detect 20% of the errors, the global detection rate can go up to 40%).

Figure 2.3 summarizes the impact of contracts on the fault localization effort. The chart evaluates the impact of contracts quality (rate of errors locally detected by each class) and their density (0,2 means that there is contract for 20% of the statements in the program). The diagnosis effort is computed as the number of statements that have to be examined before finding the error. The most important conclusion from this experiment is that the density of contracts has much less impact than their quality on the diagnosis effort.

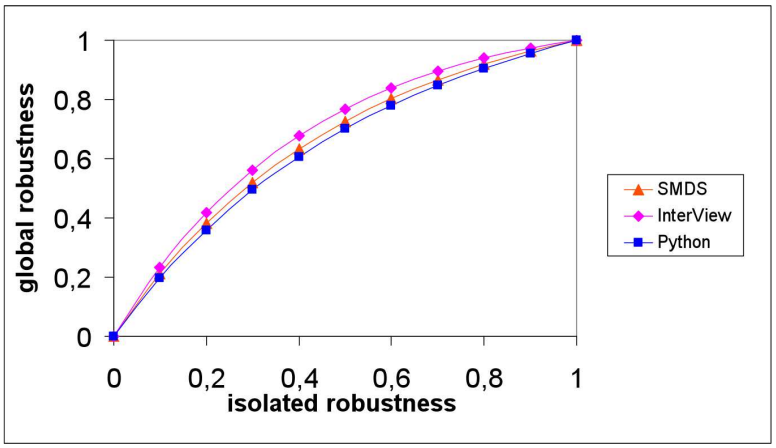

Figure 2.2: Evolution of vigilance according to the contracts efficiency

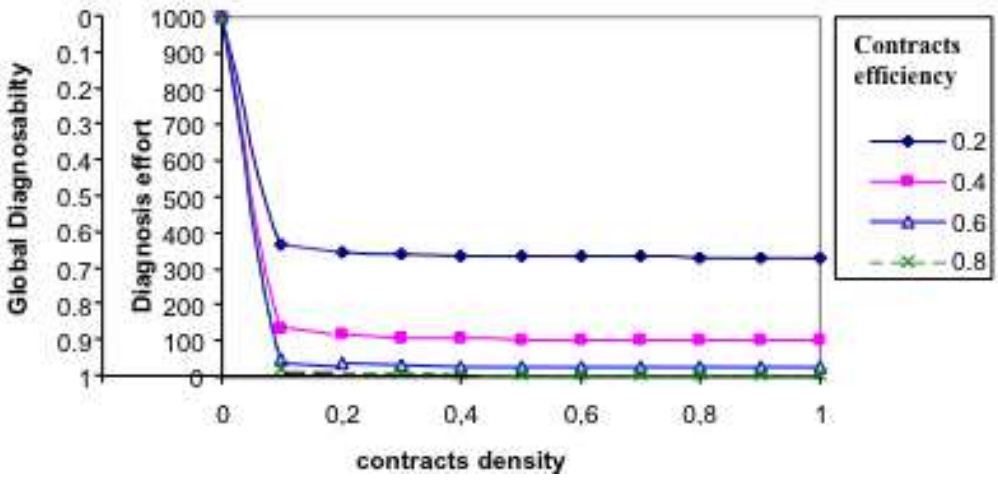

Figure 2.3: Evolution of diagnosability with contracts density

2.2 Testing contributions in the OO paradigm

This section summarizes two important contributions of our work for testing OO programs: a novel evolutionist algorithm for improving the quality of test suites; the definition of a test adequacy criterion specially designed to improve a test suite in order to improve the accuracy of trace-based fault localization algorithms.

2.2.1 The bacteriologic algorithm for automatic optimization of a test suite

The issue of automatically improving test cases is a non-linear optimization problem, and the application of genetic algorithms (GAs) looks like an interesting way to solve it. Furthermore, a strong analogy exists between natural selection and the process of generating new test cases based on an initial set of test cases. Initial test cases are of various efficiency, but each of them can participate to the optimization. The optimization problem can be modelled as follows: a test case is considered as a *predator* while a mutant program is analogous to a *prey*. The aim of the selection process is to generate test cases able to kill as many mutants as possible, starting from an initial set of predators, which is the test cases set provided by the tester.

We performed experiments with genetic algorithms to improve unit test suites for Eiffel classes and system level test suites for a C# parser in the .Net framework [100, 101]. While it was quite disappointing that these experimentation results were not as good as expected, biologists colleagues suggested to try a slight variation on this idea, no longer at the "animal" level (lions killing zebras) but at the bacteriological level. This also meant thinking about the improvement of test suites more as an adaptation problem than an evolution problem. This allowed us to propose the *bacteriologic algorithm* as an evolution of the genetic algorithm inspired by evolutionary ecology [117].

Our novel algorithm is called a bacteriologic because it is inspired by *evolutionary ecology* [117] and more particularly *bacteriologic adaptation*. Evolutionary ecology is defined as the study of living organisms within the context of their environment, with the aim of discovering how they adapt [117]. The fundamental concept of this approach is that in a heterogeneous environment it is not possible to find a single individual that fits the whole environment. It is thus necessary to reason at the population level. This actually matches the intuition for the problem we want to solve: it is not possible to generate a single "perfect" test case to kill all mutants; on the contrary a global set of test cases has to be generated and improved to kill all mutants.

The bacteriologic algorithm at a very generic level takes an initial set of bacteria and incrementally improves it in order for the set to be optimally adapted to a particular environment. New bacteria are automatically generated from existing ones through the application of a mutation function and the level of adaptation to the environment is evaluated by a fitness function. As the execution unfolds there are two test sets, the *solution set* that is being built, and the set of potential test cases, that we call a *bacteriologic medium*. The incremental optimization consists in applying the following functions on the current bacteriologic medium at each step. Let's call B the input domain

of the algorithm.

Fitness function. fitness: $2^B \rightarrow \mathbb{R}^+$

The fitness function computes a real number that evaluates the quality of the solution set. In the case of automatic test generation, this function can be based on the coverage rate of the control graph, mutation score or any other test adequacy criterion.

Memorization function. mem: B $\rightarrow$ boolean

This function evaluates if a bacterium can provide new knowledge to the solution set, based on the relative fitness of the bacterium. *relFitness:* TC $\times$ $2^B \rightarrow \mathbb{R}^+$ computes the fitness of a bacterium b (relatively to the fitness of the current solution set) as follows: relFitness(Solution, b) = fitness(Solution$\cup${b})-fitness(Solution). A bacterium is memorized in the current solution if its relative fitness is above a given memorization threshold.

Mutation function. mutate: B $\rightarrow$ B

The mutation function generates a bacterium by slightly altering an ancestor bacterium. This operator is crucial for the algorithm, since it is the one that actually creates new information in the process. The mutation rate is a variable of the algorithm that fixes the rate of bacteria in the current medium that are mutated at each generation.

Filtering function. filter: $2^B \rightarrow 2^B$

This function aims at periodically deleting useless bacteria from the bacteriologic medium to control the memory space during the execution.

For automatic test suite improvement, test cases are modelled as bacteria and we experiment with fitness functions based on mutation score and statement coverage. We have performed experiments to improve test cases for a C# parser with a genetic (figure 2.4) and a bacteriologic algorithm (figure 2.5). The charts show the evolution of the mutation score along the algorithm's execution. One can notice the following differences between the algorithms: the bacteriologic algorithm converges faster than the genetic towards a good mutation score; the bacteriologic reaches a better final score than the genetic algorithm; the bacteriologic algorithm is more stable thanks to its memorization mechanism. More details about the comparison can be found in a paper published in the Journal of Software Testing Verification and Reliability [13].

2.2.2 Reconciling test and diagnosis in OO programs

In practice, no clear continuity exists between the testing task fault and the task of locating faults in the program code. This discontinuity is evidenced by different tools and techniques: while the former aims at generating test data and oracles with a high fault-revealing power, the latter uses, when possible, all available symptoms (e.g. traces) coming from testing to locate and correct the detected faults. This discontinuity is also a usual practice in software industry: if a separate group of people is in charge of detecting errors through testing, they are usually not in charge of locating and fixing them.

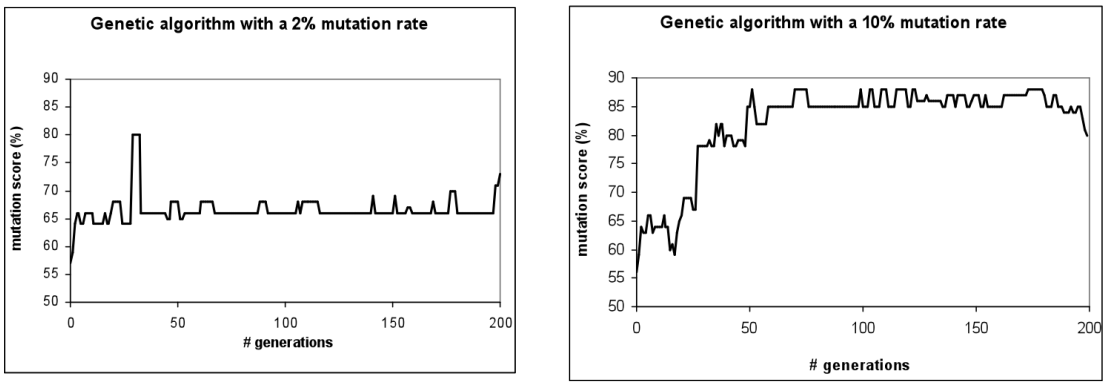

Figure 2.4: Optimizing test cases for a C# parser with a genetic algorithm

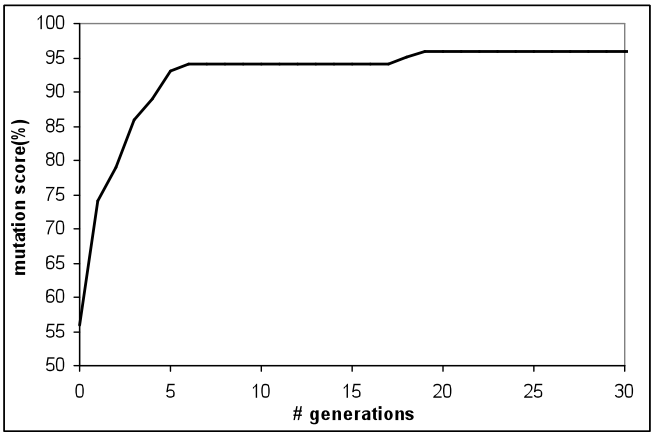

Figure 2.5: Optimizing test cases for a C# parser with a bacteriologic algorithm

However, there is a clear dependency between the two activities: the richer and more precise the information coming from testing, the more accurate the diagnosis may be. This need for testing-for-diagnosis strategies is mentioned in the literature [2, 76], but the explicit link from testing to diagnosis is rarely made. Zeller et al. [149] propose the Delta Debugging Algorithm which aims at isolating the minimal subset of test sequences which causes the failure. Delta Debugging automatically determines why a computer program fails: the failure-inducing input is isolated but fault localization in the program code is not studied.

Considering the issue of fault localization, the usual assumption states that test cases satisfying a chosen test adequacy criterion are sufficient to perform diagnosis [2]. This assumption is verified neither by specific experiments nor by intuitive considerations. Indeed, reducing the testing effort implies generating a minimal set of test cases (called a test suite) for reaching the given criterion. By contrast, an accurate diagnosis requires maximizing information coming from testing for a precise cross-checking and fault localization. For example, the good diagnosis results obtained in [76] are reached thanks to a large amount of input test data. These objectives thus seem contradictory because there is no technique to build test cases dedicated to an efficient use of diagnosis algorithms.

We address this gap through the definition of a test criterion to improve diagnosis [60, 14]. This *test-for-diagnosis criterion* (TfD) evaluates the 'fault locating power' of test cases, i.e. the capacity of test cases to help the fault localization task. An existing test suite, which reveals faults, can be improved to satisfy the TfD criterion so that diagnosis algorithms are used efficiently. The goal is to obtain a better diagnosis using a minimal number of test cases.

To define the TfD criterion we identify the main concept that reduces the diagnosis analysis effort. It is called Dynamic Basic Block (DBB) and depends both on the test data (traces) and on the software control structure. A dynamic basic block is a sequence of statements that always appear together in the executions traces generated through a test suite execution.

Definition 2. ***Test-for-Diagnosis (TfD) criterion****. A test suite satisfies the TfD criterion if it maximizes the number of dynamic basic blocks (DBB) distinguished in the program under test. A test suite will increase the number of DBBs if the execution traces generated by the test cases allow to discriminate more precisely statements one from each other in a larger number of small sequences of statements that appear together.*

The relationship between this concept and the fault localization accuracy is experimentally validated. Experimental results also validate the optimization of test suites that satisfy the TfD criterion, in comparison with coverage-based criteria. Experiments use the fault localization algorithm proposed by Jones et al. [76] and the bacteriologic algorithm [15] to automatically optimize test suites. We generate test suites with variables sizes that cover all statements, as well as test suites that satisfy the TfD criterion. We use mutation analysis [47, 110] to systematically introduce faults in programs. The efficiency of a test suite for fault localization is estimated on the seeded faults. Figure 2.6 illustrates a major benefit of the TfD criterion: it allows generating smaller test suites

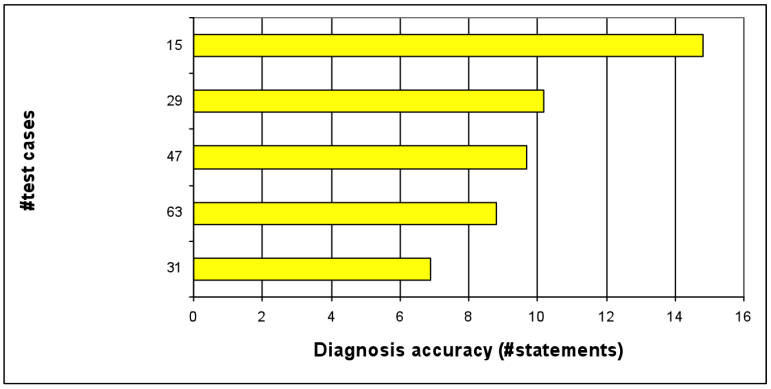

Figure 2.6: Fault localization accuracy using the TfD criterion

for a better accuracy than test suites based on statement coverage. The fault localization accuracy is computed as the number of statements that need to be inspected before finding the error. More details about this contribution to OO testing can be found in a paper published at ICSE'06 [14].

2.2.3 Tools

Most of the concepts and techniques proposed in this document have been validated through rigorous experiments. These experiments require the development of tools that implement our new ideas. In the context of our investigation of the object-oriented paradigm we have developed tools for mutation analysis in Java, C# and Eiffel and for the automatic improvement of test suites for programs developed in these three languages.

All mutation tools share a common architecture that can be used to inject errors at unit or system levels. Figure 2.7 presents the generic UML model for the two variants (unit/system) of the mutation tool. For unit testing, the operators that implement the UnitTesting interface (all) are applied, in the same way, for system testing operators that implement SystemTesting interface (LOR and NOR) are available.

We develop a Java framework to run genetic and bacteriologic algorithms. The global processes for each algorithm are developed on the basis of an abstract representation of a gene or a bacterium. The framework can then be specialized for a particular optimization task through the specialization of gene, bacterium and the fitness function. We used this framework to generate test cases for functional unit and system testing [13], for fault localization[14], and, in more recent work, to evaluate the robustness of adaptive systems [104] and to reorganize architectural components from Orange's information system in order to follow a service-oriented architecture [132].

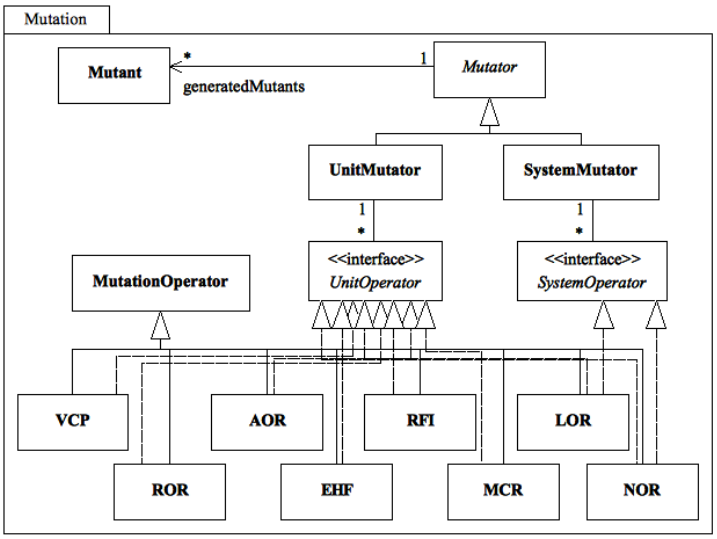

Figure 2.7: A generic model for the mutation tool

2.3 Feedback on the OO paradigm

The studies on testability of OO models led to a more precise understanding of the testing effort that can result from different decisions in an OO design. As a consequence of this study we also propose a contribution to the OO modelling activity through the definitions of additional modelling elements that can add precision about interactions in a class diagram and thus lead to a more accurate prediction of the testing effort.

2.3.1 Testability tags for UML class diagrams

Improving the testability of a UML model, with respect to our testing criterion, means either avoiding object interactions and especially concurrent accesses to shared objects, or decreasing the number of potential interactions to have a better idea of the actual testability of the design. A solution may consist in clarifying the design, so that the implementation will be as close as possible to what the designer wants. We define several stereotypes that specify the semantic of links involved in class interactions (association, dependency, aggregation, composition). These additional elements, should prevent the developer from implementing an actual object interaction that would lead to high testing costs. The stereotypes introduced here are analogous in some way to data flow testing criteria for classical software [120] that identify “definition” and “use” of variables in a program. This classical testing model aims at determining the data flow, the “life line” of variables at unit level.

Here are the four stereotypes we propose:

- «**create**»: a «**create**» stereotype on a link from class A to class B means that objects of type A calls the creation method on objects of type B.
- «**use**»: a «**use**» stereotype on a link from class A to class B means that objects of type A can call any method excluding the create one on objects of type B. It may be refined in the following stereotypes:
  - «**use_consult**»: is a specialization of «use» stereotype where the called methods do never modify attributes of the objects of type B.
  - «**use_def**»: is a specialization of «use» stereotype where at least one of the called methods may modify attributes of the objects of type B.

The absence of stereotype on a link is equivalent to a combination of «use» and «create».

Going one step further for more precise models, we study how testability stereotypes can be automatically inserted when a design pattern is instantiated in a UML class diagram. Parameterized collaborations in UML offer a promising approach to describe the structure of a pattern. In this approach, one can represent the role that should be played by a pattern participant, instead of the participant itself. In figure 2.8, we use a parameterized collaboration to represent the Abstract Factory design pattern, where two roles can be identified (Factory and Product). When this collaboration is used, i.e. each role is linked to an effective class, a dependency relation is created between these classes.

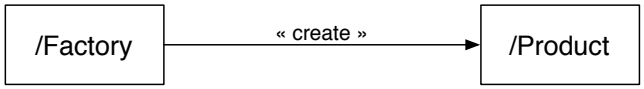

Figure 2.8: Parameterized Collaboration

However, as stated by Sunyé et al [137], collaborations still present several lacks for modelling design patterns. In this particular case, for instance, designers are not able to specify if these are the roles of individual classes or roles of class hierarchies. The designer cannot either specify that the Factory should own a "creator" method. Moreover, roles are limited to classes and associations, whereas patterns are also composed by attributes and methods.

A possible workaround for these lacks is to use collaborations to extend the UML meta-model. The idea is that pattern constraints may be attached at this meta-level so that the right stereotypes will be automatically generated each time a designer instantiates a pattern. We illustrate this approach on the Factory design pattern.

Figure 2.9 presents the same pattern, using the notation, proposed in [85], where patterns are described as meta-model level collaborations, completed by constraints. In this representation, the pattern is composed of three roles: Factory, Product and Creator. The first role is a «hierarchy», i.e. a set of classes linked by a generalization relationship. The second one is a set of hierarchies.

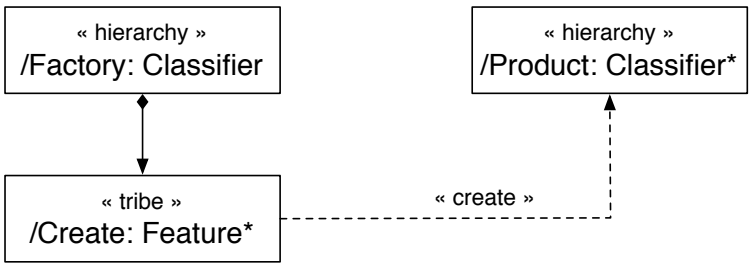

Figure 2.9: The Abstract Factory Design Pattern

2.4 Conclusion

We investigated testing for object-oriented programming in a time where on one hand the testing community was still skeptic about object-orientation while on the other hand the OO community was maturing enough to identify design patterns that are now standards [66], to propose the unified modelling language and to introduce test-driven development [25]. Thus, exploring both testing and object-oriented designs was a great experience because the relationship between both still had to be clarified and numerous questions emerged to understand how they could fit one with the other. Out of these work, I identify two major contributions:

- the bacteriologic algorithm was initially developed as an adaptation of genetic algorithms to improve object-oriented test suites. Since then, it has been used by other groups to generate test cases [84], but it has also been experimented in fields as divers as service-oriented architecture [132] or routing optimization in wireless networks [147].
- the experiments that evaluated the relationship between test and Design by Contract ©[17, 86] offered an original perspective at the intersection of OO design and test. This work has since then inspired other testing techniques [35, 38]

Chapter 3

Aspect-oriented programming

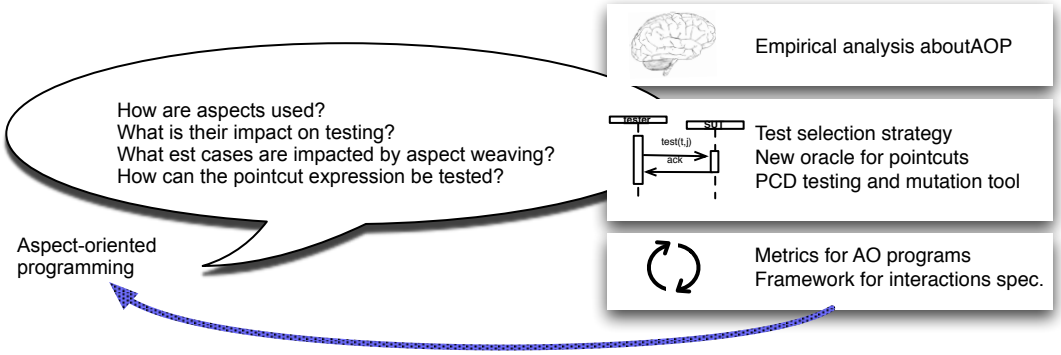

Figure 3.1: Investigating aspect-oriented programming

Object-orientation pushes forward ideas such as modularity, abstraction, and encapsulation [93]. It promotes the separation of concerns as a cornerstone to improve the maintainability, evolution, and comprehension of a software system. Since a modular unit encapsulates the behavior of a single concern, its maintenance and evolution should require modifying a single module. This results in a major improvement in comparison to non-modular design, which requires modifying several pieces of code several times [70, 8].

However, separation of concerns and modularity cannot always be achieved with OO. Some concerns cannot be neatly separated into objects, and hence, they are scattered across several modules in the software system. Such concerns are referred as *crosscutting concerns* because they are realized by fragments of code that bear identical behavior across several modules. Maintaining a crosscutting concern means modifying each fragment of the scattered code realizing that concern. Therefore, increasing the coding time, the maintenance cost and error proneness as analyzed by Eaddy et al. [51].

Aspect oriented programming (AOP) appeared in 1997 as a mean to cope with this

problem [78]. The idea underlying AOP is to encapsulate the crosscutting behavior into modular units called *aspects*. These units are composed of *advices* that realize the crosscutting behavior, and *pointcut descriptors*, which designate the points in the program where the advices are inserted. The expressive features provided by aspect-oriented languages were meant to enable developers to encapsulate tangled code in a very versatile way; therefore improving maintainability of the system by allowing the evolution of single units instead of scattered code fragments.

From our tester's point of view several question arise about aspect-oriented programming. Our first interrogations are related to our intuition that the use of declarative pointcut descriptors (PCD) might be a difficulty when analyzing and understanding the interactions between the aspects and the base program. Thus we wonder: how is the PCD used, how can it affect the correctness of the global program or how can it be tested? Other interrogations come from the fact that the introduction of aspects can dramatically change the program's behavior. Thus we wonder: how much of the invasiveness abilities of aspect-oriented programming languages is actually used and how can it be controlled?

In order to answer these questions, in section 3.1, we analyze open-source aspect-oriented programs [108]. Then, based on our understanding of the usage of aspect-oriented mechanisms we develop some testing techniques that consider specificities of AOP: a static test selection process after aspect weaving (section 3.2.1) and an automated oracle mechanism for pointcut descriptors 3.2.2. We also have two contributions to the domain of aspect-oriented programming, detailed in section 3.3: a framework for the specification and the verification of interactions between aspects and base code; a metrics suite for aspect-oriented programs. As shown in figure 3.1, our investigation of aspect-oriented programming led to contributions on the whole spectrum of the QLTF pattern. This work has been developed mainly as part of Romain Delamare's PhD thesis [41] and Freddy Munoz' Master's thesis.

3.1 What we learned about aspect-oriented programming

Since its introduction in 1997, many technical documents, research papers, books, and conference venues discussed and commented on AOP and its benefits. In 2001 the MIT announced AOP as a key technology for the future 10 years [143]. Later, in 2002 a growing scientific community launched the first International Conference on Aspect Oriented Software Development (AOSD), and about 300 (according to an estimation using the Google scholar © search engine) documents cited AspectJ and AOP. The same year less than 10 open-source projects were actually using such technology in the sourceforge repository.

Nowadays, 8 years after the MIT announcement, the number of documents about AOP and AspectJ has grown to more than 2500. During the same period, the number of projects using AOP has increased only to about 60 projects (less than 0.5% of sourceforge's projects developed using Java in the period from 2001 to 2008 integrate aspects). When facing this apparent paradox, we can wonder what prevents a more extensive use

of AOP in what context it has been a good solution.

Previous work has identified two characteristics of aspect-oriented languages that hinder maintainability and evolvability: (1) the fragility of the pointcut descriptors that leads to the evolution paradox [141]; (2) the ability of aspects to break the object-oriented encapsulation [106]. Also, when looking at aspects for analysis or testing, another paradox seems to occur: aspects allow the extraction of scattered code in a single unit, thus improving the consistency of modules, but aspects can also increase coupling between modules when woven at multiple places. This increased coupling has a negative impact on testability, since it prevents an incremental approach for testing. In turn, this decreases maintainability because the testing effort is impacted each time the program evolves.

We perform a two-step empirical analysis of AOP in order to understand more precisely how aspects are used [108] and what is their impact on program quality. First, we analyze the current usage of aspects features in open source projects. We study 38 open source aspect-oriented projects developed with the Java and AspectJ languages and ask the following questions about usage:

- What is the extent of aspect and invasive aspect usage in AO projects? Does this usage vary with the size of the project?
- To what extent do aspects and invasive aspects really crosscut AO systems? Does this depend on the size of the systems?
- Do pointcut descriptors use the full expressivity provided by the AspectJ pointcut language? Are invasive advices woven with precise pointcuts?

Our observations reveal that developers of open source aspect-oriented programs use few advices to modularize crosscutting concerns (the mean ratio of advices per base program method is 0,05%), and that these advices are scarcely crosscutting (70% of the advices are woven at one joinpoint only). When focusing only on invasive advices, we observe that developers write very few advices that break object-oriented encapsulation, and the small number of invasive advices, advise a small number of very specific joinpoints. The observations on the coverage of the PCD language confirm this: developers write specific PCDs using only half of the AspectJ PCD language's expressiveness.

An explanation for the slow adoption of AspectJ might reside on its inability to hold its promise of improved maintainability. In order to go further in understanding this slow adoption of AspectJ, we evaluate its impact on a particular dimension of maintainability: testability. Our hypothesis for the second part of the empirical analysis is that developers do not use AspectJ because aspects introduce more difficulties to test than regular Java classes. We compare the evolution of testability indicators over 3 versions of a system implemented with both Java and AspectJ technologies. This reveals that in the AspectJ versions, modules are more cohesive but are also more coupled. The increased coupling among modules suggests that AspectJ reduces testability by introducing modules that cannot be tested in isolation. More details about this work can be found in a paper published at ICSM'09 [108].

3.2 Testing contributions for aspect-oriented programming

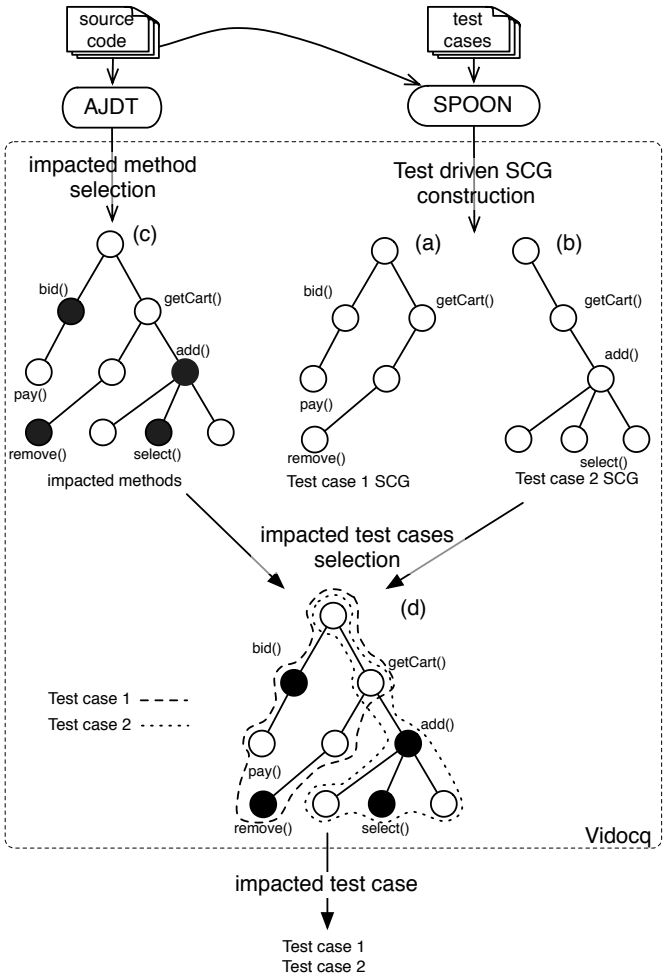

Figure 3.2: Global analysis process for test selection

3.2.1 Static test selection after aspect weaving

Weaving aspects in a program impacts its existing behavior. If test cases exist for the program, we must identify the ones that validate impacted behaviors to check the interactions between the program and the aspects. The test selection problem consists in identifying which test cases have to be modified and re-executed after aspect weaving (and which new test cases must be generated). Several approaches exist [148, 150] which allow selecting only the impacted test cases (e.g. executing at least one joinpoint in the program). These dynamic techniques require the execution of all test cases and

the instrumentation of the program before and after evolution.

We propose a static analysis performed in a two steps process [44]. First, the analysis leverages the pointcut descriptor that identifies all the points in the program that are impacted by the aspect weaving. Then, the test cases and the base program are analyzed in order to determine the points reached by the test cases. If a test case reaches a point that is impacted by an aspect, it is identified as an impacted test case. From the user point of view, the advantage of such a static analysis is the fact that the analysis is instantaneous, compared to an approach for which the execution traces of test cases on the base program must be exploited using code instrumentation.

Because our analysis is static, it over approximates the set of impacted points in two cases. First, a pointcut can declare a dynamic expression. In that case, we statically over approximate the set of joinpoints that can be matched by the aspect. Second, we have to over approximate the coverage of test cases in case of a method call on a polymorphic object, since it is not always possible to statically know the type of the object. This potential threat to the validity of the approach has to be estimated and qualified. Although the algorithms for the analysis are independent of any technology, we had to make choices for the tool. We chose to build Vidock for AspectJ aspects and JUnit test cases [46]. Because of the over approximations discussed earlier, we are able to completely analyze the AspectJ and Java languages.

Figure 3.2 presents an overview of the analysis process. Inputs are a set of test cases and AspectJ source code. Vidock calls the AJDT [52] to statically analyze the aspects, the base code and identify the methods in which at least one aspect will be woven (c). Spoon [114] is used to analyze the JUnit test cases. Then, it generates a static call graph (SCG) for each test case (a,b). Crossing both analyses, Vidock selects the impacted test cases by identifying intersections between the set impacted methods and methods in the SCG (d).

We have experimented Vidock on 5 systems. In most cases the aspects impact only a few test cases. We also observe that over-approximations have a minimal effect on the results. We have also measured the occurences of dynamic pointuts and method overriding in 46 open-source AspectJ projects. We observe that 65% of the projects can compile without aspects, and thus the base program can be tested in isolation; 6% of the methods are overridden and 17% of the pointcuts are dynamic, so the effect of over-approximations should be reasonable in general.

3.2.2 An oracle for AspectJ pointcut descriptors

The development of testing techniques that are well suited for AOP requires considering features unique to aspect oriented languages. A number of researchers have studied the new types of faults that can be introduced by AOP: faults that can occur in the interactions between the core concerns and the aspects, in the advice, or in the pointcut descriptor (PCD). The latter category of faults is specific to aspect-oriented languages that introduce new constructs to define the PCD. As observed by Ferrari et al. [55], the PCD is the most error-prone part of an aspect.

We focus on the definition of an oracle for the PCD, in association with a tool and method to assist in the definition and validation of pointcut descriptors. An incorrect PCD can generate numerous faults in the program: it can miss expected joinpoints and / or match unintended joinpoints. As a consequence, the advice can introduce unexpected behavior at numerous places. Alternatively, expected behavior can be missing at several places in the program. This is known as the fragile pointcut problem in AOP [136]. Unforeseen problems can also occur when evolving aspect-oriented programs [107]. There is a well-known risk that the PCD may match unintended joinpoints. This is known as the evolution paradox issue in AOP [141]. Pointcut descriptors thus represent an important issue for the correctness of aspect-oriented programs.

A major challenge to detecting faults in the PCD is that it declares very specific joinpoints and there is no other specification of the set of joinpoints that it should match. This means there is no simple way to define an oracle for a test case that looks for errors in a PCD. For example, when developing an aspect-oriented program with AspectJ, there exists no testing approach that allows a tester to specify the expected locations of joinpoints. One way to address this problem is to write test cases, such as with JUnit, that check whether or not the behavior introduced by the advice executes correctly at the expected place in the program. The drawback of this solution is that these test cases do not target the correct type of fault: they target faults in the advice's behavior, and, only as a side effect, may capture faults in the PCD, such as if the advice does not execute correctly because it has not been woven at the expected joinpoint.

We propose to monitor the execution of advices and define a set of query operations to retrieve information about this execution [43, 42]. For example, one operation can check the presence of an advice at specific place in the execution of the program. Such operations are meant to build automatic oracles for PCD test cases. We have developed a tool called AdviceTracer, which handles monitoring and storage of information regarding what advices defined in a particular aspect are executed and at which place in the base program. AdviceTracer offers a set of operations to retrieve this information. These operations can be used to define test cases that specifically target the presence or absence of an advice. AdviceTracer can be integrated with existing testing frameworks such as JUnit in order to develop test cases for PCDs. We propose an approach for testing PCDs based on AdviceTracer to capture faults that miss joinpoints or that match unintended joinpoints.

We evaluate AdviceTracer in terms of its usability and utility for writing test cases that target faults in AspectJ PCDs. The study is performed as a comparison between test cases that use JUnit only and test cases that use AdviceTracer with JUnit. We use the Healthwatcher system that has 93 classes and 19 PCDs. Healthwatcher is a popular benchmark and has been used in several research studies on aspect-oriented software development.

The empirical study considers two questions. First, we evaluate the ability of AdviceTracer to decrease the effort for writing the test cases. We evaluate this effort through the number of test cases, the complexity of the test cases, and the precision of the oracle. The second question evaluates the ability of AdviceTracer to improve the fault revealing

abilities of test cases. Here, we perform a mutation analysis and compare the number of mutants killed by each approach, the time taken to kill them, and the number of equivalent mutants detected.

Figure 3.3 synthesizes the results of this study. The tester who used AdviceTracer took less time than the one who used JUnit to complete the testing of all PCDs in HealthWatcher. The tester with AdviceTracer was also able to develop test cases that detect more errors. The sizes of the two test sets are comparable, but the test set that uses AdviceTracer has much less lines of code. This is because AdviceTracer proposes an automatic oracle mechanism that is much more precise for PCDs than JUnit: AdviceTracer allows to express assertions directly about the expected behaviour of PCDs, while JUnit only allows to express assertions about the expected behaviour of the program in which aspects have been woven. As a consequence, test cases developed with AdviceTracer use shorter test scenario and more compact oracles.

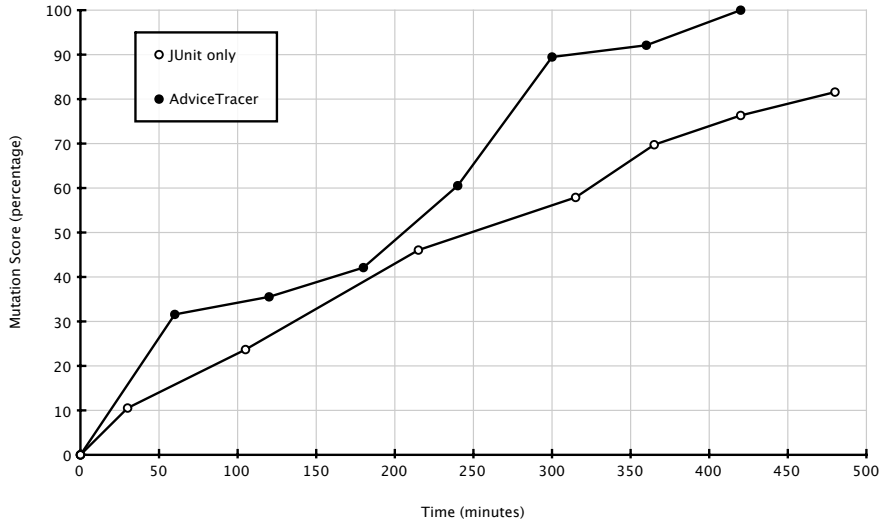

Figure 3.3: Comparing AdviceTracer and JUnit for error detection in PCDs

More details about AdviceTracer and associate experiments can be found in [43, 42].

3.2.3 PCD mutation tool

We developed the AjMutator mutation tool [45] as part of our experiments for testing AspectJ pointcut descriptors. AjMutator can generate mutants by introducing faults in the PCDs, and execute a set of test cases on the generated mutants. 4 summarizes the set of mutation operators that AjMutator can apply on AspectJ pointcut descriptors. AjMutator also automatically classifies the mutants by comparing the sets of joinpoints matched by the mutant and the initial PCD. We automate this classification at compile time by leveraging the static analysis performed by the compiler that computes the set

Operator	Description
PCCC	Replaces a cflow PCD with a cflowbelow PCD, or the contrary
PCCE	Replaces a call PCD with an execution PCD, or the contrary
PCGS	Replaces a get PCD with a set PCD, or the contrary
PCLO	Changes the logical operators in a composition of PCDs
PCTT	Replaces a this PCD with a target PCD, or the contrary
POEC	Adds, removes or changes exception throwing clauses
POPL	Changes the parameter list of primary PCDs
PSWR	Removes wildcards
PWAR	Removes annotation from type, field method and constructor patterns
PWIW	Adds wildcards

Table 3.1: AjMutator mutation operators for pointcut descriptors

of joinpoints statically matched by the PCDs. We show that a benefit of this classification is that, for a class of PCDs, we can conclude that if the mutant matches the same set of joinpoints as the initial PCD, the mutant is equivalent.

3.3 Feedback on aspect-oriented programming

Looking at aspect-oriented programming and AspectJ in particular, it appears that it is very difficult to understand the interactions between aspects and base code. This difficulty implies a risk for errors and problems for locating these errors. Our first feedback to AOP is a framework called ABIS in which it is possible to declare and check expected interactions between aspects and base program in AspectJ.

Our empirical analysis of usage and testability of aspect-oriented programs relies on the computation of a set of metrics that can reveal trends about AOP. We developed a framework to model any aspect-oriented program and defined the metrics on the basis of this framework. This measurement environment is our second contribution as a feedback to AOP.

3.3.1 ABIS: a framework for aspect interaction specification and verification

Invasive AOP approaches use composition mechanisms that allow developers to manipulate almost any structure of the base program. This ability to manipulate the base program structures is called invasiveness. Invasive AOP provides several strategies to manipulate the base program. These strategies range from less invasive such as the augmentation of a procedure execution to more invasive ones such as the replacement of a procedure execution. An *invasiveness pattern* is the characterization of invasive behavior, i.e. the strategy or a combination of them to manipulate the base program.

Invasive aspects are useful to introduce functionalities that otherwise must be hard-coded into the base program. For example, a system transaction concern is implemented using an invasive aspect because it requires stopping the execution of the intercepted behavior each time a transaction fails. However, invasive aspects can also do harm to the base program. When they introduce the functionalities they are designed for, they can also introduce side effects, hence, generating unexpected interactions.

We have precisely illustrated the issues that can arise when evolving an aspect-oriented program that is built using invasive aspects [107]. This is a special case of the AOSD-Evolution paradox [141], which results to be aggravated in presence of invasive aspects. We consider a distributed chat application as a case study. This study demonstrates the need to reason about expected interactions, and to control the usage of invasive aspects.

In the second part of this work we propose a framework for specifying the expected interactions in the base program as well as the way in which aspects can be invasive. The specification on aspects is based on a classification proposed in previous work [107]. This classification identifies 11 different patterns according to which AspectJ aspects can be invasive. Based on this specification, the base program declares which type of invasiveness it allows or forbids: Augmentation, Replacement, Conditional replacement, Multiple, Crossing, Write, Read, Argument passing, Hierarchy, Field addition, Operation addition.

We have developed the ABIS tool that supports this framework. ABIS has several features to support the analysis of interactions between aspects and base code:

- analyze aspects to automatically infer which invasive pattern they encapsulate
- an annotation-based language to declare the patterns an aspect implements and the patterns the based code can accept. Figure 3.4 shows an example of the declaration of the FlowConditionReplacement pattern for an aspect and figure 3.5 shows the declaration of a forbidden pattern in the base code (it is not possible to weave an aspect that implements the FlowConditionReplacement pattern on the move method)
- a static checker to validate that aspects conform to the specification of the base program.

Experiments have shown that specifying interactions is useful to early detect when invasive aspect can perform harmful. ABIS statically computes and gives information, at compile-time, about the specification violation. This information is a useful and valuable:

- Feedback for developers in the process of writing advices a specifying the base program
- For verifying an aspect-oriented program when aspects and the base program are developed separately

```
@FlowConditionReplacement

void around(Object obj,List list):execution(* add(Object))&&
       args(obj)&&target(list){
       if(obj==null){
              System.err.println("Null elements cannot be added");
              return;
       }
       else proceed(obj,list);
}
```

Figure 3.4: Pattern specification on an aspect

```
public class Glass {
       public int x, y;

       @CMForbidden(forbid={AspectClassificationType.
                            FlowConditionReplacement})
       public void move(int dx, int dy){
              x += dx;
              y += dy;
       }
}
```

Figure 3.5: Example for forbidden pattern declaration in base program

- For verifying an aspect-oriented program when aspects or the base program evolve.

More details about this work have been published at ICSM'08 [107].

3.3.2 A framework for the definition of AO metrics

Briand et al. [34] proposed a formalism for analyzing object-oriented programs. The goal of Briand's work was to define a terminology and a formalism to capture the core concepts of object-oriented programming independently of any language. This formalism then allowed the definition of generic metrics for object-oriented programs.

We extend Briand's framework to support aspect-oriented concepts. Our goal with this extension is to provide the support for aspect-oriented metrics independently of a particular aspect-oriented programming language. The benefit is that the we can reuse the metrics definition to measure programs implemented with AspectJ, CeasarJ or JBoss AOP for example, as illustrated in figure 3.6. In order to measure metrics on a specific implementation it is necessary to extract a model from the program that contains all the concepts defined in our theoretical framework. Then the metrics definitions can be reused. We keep everything that was in Briand's framework, since aspect-orientation is an extension of object-oriented programming, and we adapt the definitions to aspect-oriented programming. This also guarantees that, if an object-oriented program that

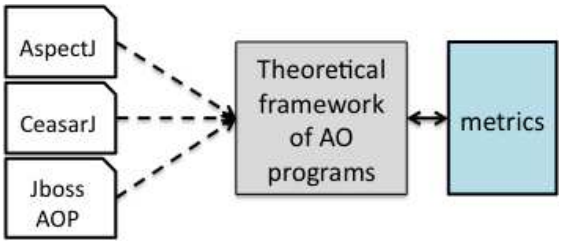

Figure 3.6: A generic metrics framework for aspect-oriented programs

has no aspects is analyzed, we can still gather the OO metrics. We also define additional concepts that exist only in the aspect-oriented paradigm. Informally, the framework captures the fact that aspect-oriented programs consist in a set of classes, methods, attributes. It also defines the notion of invocation of methods and introduces the specific concept of advice that is a method woven in some classes. It has to be noted that the notion of pointcut descriptor does not explicitly appear in the framework since it is not a fundamental structural feature of an AO program and it takes various different forms depending on the AOP language.

A number of metrics for aspect-oriented programs can then be defined on the basis of this generic representation of the program, independently from a specific language. Some metrics simply count elements (*e.g.*, number of methods, number of advices, etc.) and some metrics perform some computation on the model (*e.g.*, lack of cohesion, coupling, etc.).

In order to extract a model and compute the metrics on AspectJ programs, we have developed a set of tools. It can measure both generic AO metrics and AspectJ specific metrics on an AspectJ program. The major components of the tool are illustrated in figure 3.7. The tool is based on AJDT [52] to statically extract an abstract syntax tree (AST) from the AspectJ program. Starting from the AST we build an abstract aspect model (AAM) of the program. This model contains all the elements defined by the theoretical framework. Then, once the model is built, it is possible to use the generic metrics definitions to compute all generic AO metrics.

This analysis is performed by two different modules: the ABIS tool [107], which extracts all information related to interaction patterns in AspectJ programs; the AJExtractor module, which extracts all information about methods, invocations, attributes and computes the Uses predicate. It is worth mentioning that this module can additionally compute the AspectJ specific metrics directly on the AST.

3.4 Conclusion

This investigation of testing techniques for aspect-oriented programming is characterized by a strong emphasis on the *Learn* facet of the QLTF pattern. We started this work with very mixed feelings: on one hand there was a huge interest in AOP that generated

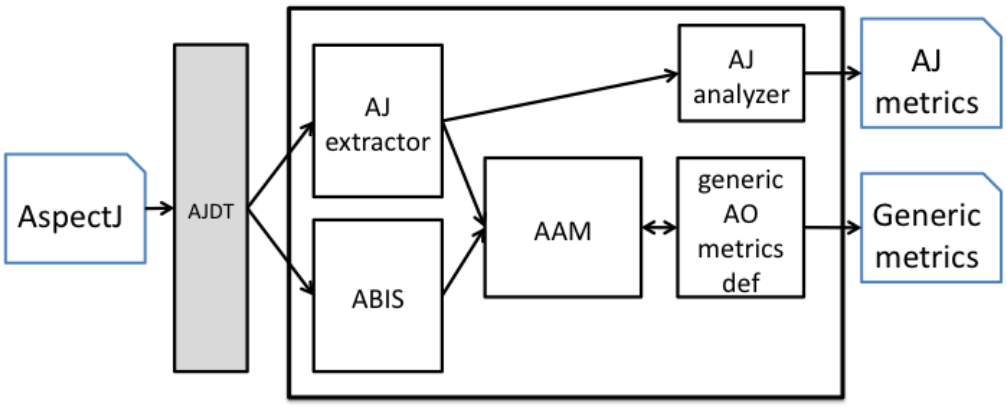

Figure 3.7: A tool for measuring AspectJ programs

innovative work in language design, code analysis, modularity, but also severe criticism against this new paradigm [135]; on the other hand, from the testing perspective, it seemed there was nothing new. But most of all, it seemed that AOP was very difficult to control and prone to complex errors deriving from the use of the advanced language features developed in aspect-oriented languages. Thus, we needed to understand how AOP could work, how it was used and what type of new interactions it could introduce in programs. This led us to empirically analyze a large set of aspect-oriented program and propose new insights on AOP and how AspectJ features are in open source project [108]. To my knowledge, this is the most extensive study on aspect-oriented programming.

Chapter 4

Model transformation

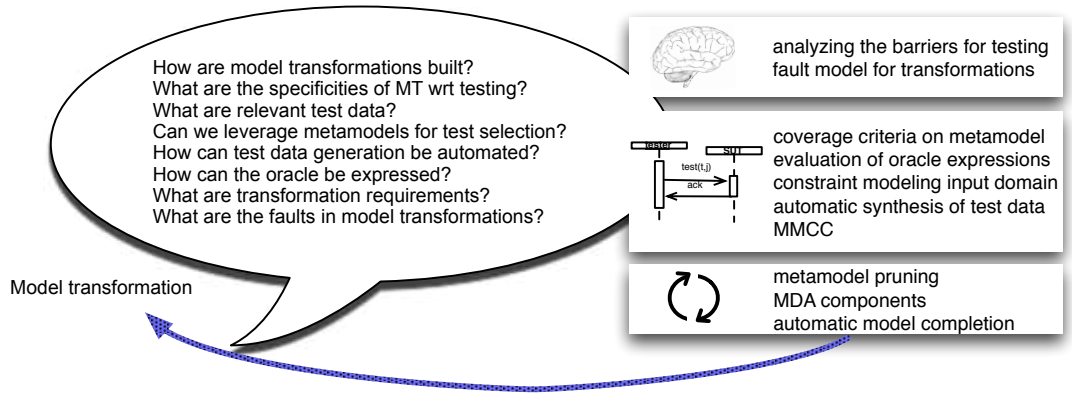

Figure 4.1: Investigating model transformation

Model Driven Engineering (MDE) techniques [123] support extensive use of models in order to manage the increasing complexity of software systems. Appropriate abstractions of software system elements can ease reasoning and understanding and thus limit the risk of errors in large systems. Automatic model transformations play a critical role in MDE since they automate complex, tedious, error-prone, and recurrent software development tasks. Airbus uses automatic code synthesis from SCADE models to generate the code for embedded controllers in the Airbus A380. Commercial tools for model transformations exist. Together from Borland is a tool that can automatically add design patterns in a UML class model, Esterel Technologies have a tool for automatic code synthesis for safety critical systems.

Other examples of transformations are refinement of a design model by adding details pertaining to a particular target platform, refactoring a model by changing its structure to enhance design quality, or reverse engineering code to obtain an abstract model. These software development tasks are critical and thus the model transformations that

automate them must be validated.

A simple example of a model transformation consists of *flattening* a state machine. Figure 4.2 illustrates a transformation that takes a hierarchical state machine as an input and produces a flattened state machine as an output. The output is semantically equivalent to the input but all hierarchical states are removed.

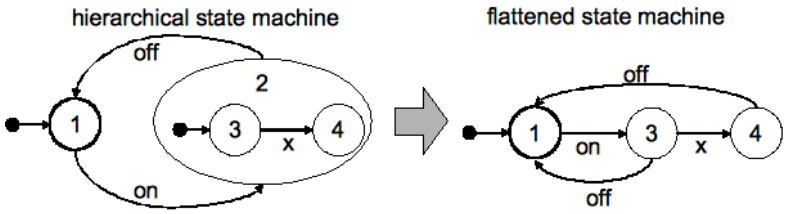

Figure 4.2: Model Transformation Example: Flattening a Hierarchical State Machine

The sets of possible input and output models for a transformation are each described by a metamodel. A metamodel is a set of classes, relationships, and multiplicities that define the concepts and the structure of a modelling language. Figure 4.3 shows a metamodel for the structure of hierarchical state machines. A hierarchical state machine contains states that have a number of incoming and outgoing transitions. The metamodel distinguishes between two types of states, simple states that can be initial or final states, and composite states that can contain composite or simple states. This metamodel is a formal description of the structure of statecharts.

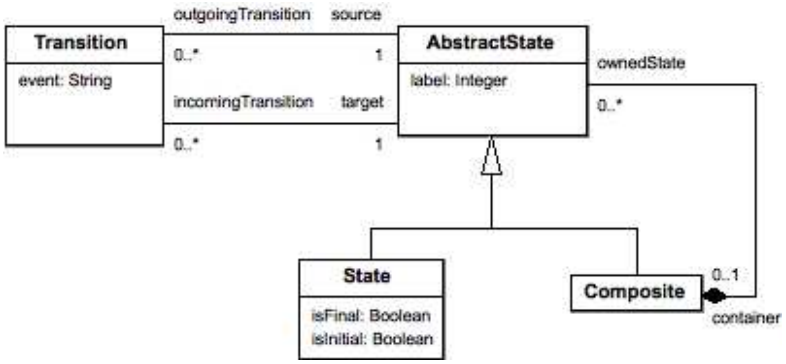

Figure 4.3: A Hierarchical State Machine Metamodel

In addition to describing metamodels with classes, attributes, and associations, it is usually necessary to define constraints that relate the concepts in the metamodel more precisely. For example, the metamodel shown in figure 4.3 does not constrain a composite state to contain only one initial state. This constraint must be added to the metamodel but it cannot be defined using modelling languages such as EMOF. The Object Constraint

Language (OCL) is commonly used to define these additional constraints. The constraint on the composite state could be written as follows:

```
context: Composite

inv: self.ownedState ->
  select(AbstractState as | as.oclIsTypeOf(State)) ->
    select(AbstractState s | s.oclAsType(State).isInitial) ->
      size()=1
```

Listing 4.1: A constraint that forbids multiple initial states

A fault in a transformation can introduce a fault in the transformed model, which if undetected and not removed, can propagate to other models in successive development steps. As a fault propagates further, it becomes more difficult to detect and isolate. Since model transformations are meant to be reused, faults present in them may result in many faulty models.

To test the transformation shown in figure 4.2, we must perform the following activities:

- Generate test data: We need to generate input models that conform to the input metamodel of the transformation. The input models that are generated as inputs for a transformation are called test models. In our example, this requires generating several state machines that conform to the metamodel of figure 4.3. Test models are manually or automatically generated in the form of graphs of metamodel instances.

- Define test adequacy criteria: Since it is clearly not possible to test a transformation with all possible input models, we must define criteria to efficiently and effectively select test models. Test adequacy criteria drive the selection of a subset of test models that will be sufficient for testing. This helps reduce the time and effort spent on testing. In our example, a test criterion for the flattening transformation could require that each class of the metamodel is instantiated in at least one test model. No well-defined criteria exist for model transformation testing. This challenge must be tackled to make testing practical and systematic.

- Construct an oracle: For software testing, the oracle determines if the result of a test case is correct. There are several ways to construct an oracle for a model transformation. If the expected model is available (e.g., from previous regression tests), the oracle can compare the output model with the expected model. If the expected model is not available (as is usually the case), a partial oracle that checks expected properties of the output model should be constructed. For example, such an oracle can check that the output model contains no composite states and has the same number of simple states as the input model.

Model transformations constitute a class of programs with unique characteristics that make testing them challenging. The complexity of input and output data, lack of model management tools, and the heterogeneity of transformation languages pose special problems to testers of transformations. In section 4.1, we identify current model transformation characteristics that contribute to the difficulty of systematically testing transformations [16]. On the basis of this survey of challenges, section 4.2 introduces specific testing techniques [16, 126] and section 4.3 proposes feedback to MDE with some techniques that lack for testing [130, 129]. All this work has been developed as part of Franck Fleurey's [57], Jean-Marie Mottu's [96] and Sagar Sen's [125] Phd theses.

4.1 What we learned about model transformation

We have focused our understanding of model transformation on two points: learn about the way transformations are developed in order to understand the difficulties for testing these particular programs [16]; fault models for model transformations in order to precisely understand what a testing technique should look for [97].

4.1.1 Barriers to Systematic Model Transformation Testing

When surveying the development methods for model transformation, we identify three major difficulties for testing: the complexity of input and output data, the lack of model management tools, and the heterogeneity of transformation languages pose special problems to testers of transformations.

Complex Input and Output Data

Model transformations manipulate data of a complex nature. The input and output models are graphs of objects that are often large. Sometimes, the input or output models contain multiple views (e.g., UML class diagram and sequence diagram views), and thus consistency of views manipulated by transformations becomes a concern.

The structure of the graphs is constrained by a metamodel and the objects in the graphs are instances of the classes defined in the metamodel. The metamodels can themselves be large and complex structures. Moreover, additional constraints can be expressed in the metamodel, usually with the OCL. This increases the complexity of the metamodels; OCL is a rich language with which it is possible to define complex constraints relating a large number of elements in the metamodel.

The complexity of the data manipulated by a transformation affects the generation of test models. Manual test generation is error-prone because of the large number of metamodel instances that must be created, and the relationships and attribute values that must be set. Automatic test data generation is a complex constraint solving problem because it requires synthesizing a graph that satisfies a large set of multiplicity and OCL constraints, and test adequacy criteria. The main challenge for automatic resolution of

complex constraints is handling time and memory when exploring very large solution spaces.

Since the metamodel completely describes the input domain of a transformation, it provides a basis for defining test adequacy criteria. It is possible to define a large number of criteria, such as instantiate all the meta-classes, combine different values for the properties, and combine instances of different meta-classes. However, lack of historical data on the types of errors typically found in transformations makes it difficult to determine the effectiveness of these criteria and the fault models they can target.

The complexity of the output data complicates the oracle problem. It is difficult to manually or automatically build the expected test result. When the expected output model is available, the oracle needs to compare two models, i.e., two graphs of objects. In this case, the oracle problem is as complex as the graph isomorphism problem, which is NP-complete. If the oracle is specified by listing expected properties of the output model, the construction of this oracle is complicated by the complexity of the output metamodel that describes the output model. The tester must consider numerous concepts and relationships to define the expected properties of the output model.

Model Management Environments

MDE development environments lack adequate support for model manipulation [65]. For testing model transformations, support is needed for building, editing, visualizing, and analyzing models.

The construction of models involves either writing a program that builds the metaclass instances and sets all the properties, or using model editors generated from a metamodel (e.g., the default tree editor generated by EMF) to manually build the instances and set values for all attributes and references. Writing a program is error-prone. Using a generated editor instead of a tailor-made editor for a language is tedious because they do not provide language-specific icons, dialog boxes for setting attributes values or assistance for checking the completeness of the model (e.g., all attributes have been assigned a value). This makes the manual definition of test models difficult and error-prone.

Visualizing output models is difficult because graphical editors (e.g., for UML or domain-specific languages) often do not provide adequate support for layout of diagrams that are produced by a transformation or exported from another tool. A confusing layout complicates manual analysis of the model, and the visual comparison of two graphical representations. A possible solution is to query the output model and check some properties using OCL analyzers.

For regression testing, testers need to compare the output models produced by two versions of the transformation because a test model that was used in testing a previous version of the transformation should produce the same output model. Thus, we need sophisticated model comparison tools.

Heterogeneity of Transformation Languages and Techniques

The OMG has defined a model transformation standard called QVT. However, there exist a large number of model transformation languages and techniques. Transformations can be implemented with general purpose programming languages (e.g., Java) or languages dedicated to model transformations (e.g., Query/View/Transformation – QVT)). The MTIP workshop [31] organized at MoDELS'05 was concerned with developing "an increased understanding of the relative merits of different model transformation techniques and approaches". The workshop organizers specified several model transformations and asked the authors to implement them. Eight papers presented 13 different techniques, which were divided into three categories**:** graph transformation related approaches, declarative and rule based approaches, imperative and related approaches. Moreover, there are several tool-specific model transformation languages, such as MetaEdit+ [91], and XMF-Mosaic. The approaches presented in the workshop reflected the diversity of existing transformation languages and techniques.

Since it is impossible to know if any one of the many techniques will fit all the needs for model transformations, the testing techniques need to take this diversity into account. The diversity has a strong impact on the definition and the selection of effective white-box test adequacy criteria. We cannot choose one language as a reference and develop test criteria that are based on language elements.

4.1.2 Fault models

All faults that can occur in object-oriented or rule-based, declarative programs can also occur in model transformations programs. However, in order to precisely understand what a test case should detect in a model transformation, the usual fault models are not sufficient. First, because usual fault models do not capture the specific faults transformation programmers may do if they are competent. They may forget some particular cases (e.g. forget to deal with the case of multiple inheritances in an input model), manipulate the wrong model elements etc. Since erroneous model transformation will differ from the correct one by complicated modifications in the transformation, fault models cannot consider a single faulty statement. Second, since a transformation program navigates both the input and the output metamodels, most simple faults will disturb this navigation in a non-consistent way (e.g. trying to navigate non-existing association due to a syntactic replacement). Thus, these faults will be detected either during programming, at compilation or at runtime. Third, the fault models should not take advantage of a transformation language's syntax. Indeed, today there are lots of model transformation languages which all have their specificities and which are very heterogeneous (object oriented, declarative, functional, mixed). That leads us to choose to focus on the semantic part of the transformation instead of the syntactic one imposed by a language.

To address the issues listed above, we define fault models on an abstract view of the transformation program, by answering the following question: which type of fault could be done during a model transformation implementation? For example, a transformation goes all over the input model to find the elements to be transformed, a fault can consist

in the navigation of the wrong association in the metamodel, or in selecting the wrong elements in a collection. During a transformation, output model elements have to be created; a fault can consist in creating elements with the wrong type or wrong initialization. The analysis of these possible faults for a model transformation leads to distinguish 4 abstract operations linked to the main treatments composing a model transformation:

- navigation: the model is navigated thanks to the relations defined on its input/output metamodels, and a set of elements is obtained.
- filtering: after a navigation, a set of elements is available, but a treatment may be applied only on a subset of this set. The selection of this subset is done according to a filtering property.
- output model creation: output model elements are created from extracted element(s).
- input model modification: when the output model is a modification of the input model, elements are created, deleted or modified.

These operations define a very abstract specification of transformations, which highlights the error-prone steps when programming a model transformation. Any model transformation combines and mixes these 4 operations of navigation/filtering (read mode) and output/input model modification (write mode). We thus identify possible faults for model transformations on the basis of these abstract operations [97].

Faults related to the *navigation*:

- Relation to the same class change: This fault replaces the navigation of one association toward a class with the navigation of another association to the same class.
- Relation to another class change: This fault replaces the navigation of an association toward a class with the navigation of another association to another class.
- Relation sequence modification with deletion: During the navigation, the transformation can successively navigate a sequence of relations, this fault removes the last step from the composed navigation.
- Relation sequence modification with addition: This fault is the opposite of the previous fault.

Faults related to the *filtering*:

- Collection filtering change with perturbation: This fault modifies an existing filtering, by influencing its parameters. One criterion could be a class' property or the class' type; this fault disturbs this criterion.

- Collection filtering change with deletion: This fault deletes a filter on a collection; the mutant returns the collection it was supposed to filter.
- Collection filtering change with addition: This fault is the opposite of the previous one. It uses a collection and processes a useless filtering on it.

Faults related to the *creation*:

- Class' compatible creation replacement: This fault replaces the creation of an object by the creation of an object of a compatible type. It could be an instance of a child class, of a parent class or of a class with a common parent.
- Classes' association creation deletion: This fault deletes the creation of an association between two instances.
- Classes' association creation addition: This fault adds a useless creation of a relation between two class instances of the output model, when the metamodel allows it.

More details about these fault models, including examples and illustrations, can be found in a paper published at ECMDA [97]

4.2 Testing contributions to model transformations

We start from a global roadmap for model transformation testing issues [61] in order to develop effective techniques. We detail here the definition of coverage criteria on the input domain, the automatic synthesis of test models and some experiments to express the oracle.

4.2.1 Coverage criteria on source metamodel

To test a model transformation, a tester will usually provides a set of *test models* that conform to the source metamodel of the transformation, run the transformation with these models and check the correctness of the result. While it is fairly easy to provide some input models, qualifying the relevance of these models for testing is an important challenge in the context of model transformations[16]. As for any testing task, it is important to have precise adequacy criteria that can qualify a set of test data. For example, a classical criterion to evaluate the quality of the test data regarding a program is code coverage: a set of test data is adequate if, when running the program with these data, all statements in the program are executed at least once. Other criteria are functional or "black-box" [27] and rely only on a specification of the system (input domain or behavior) under test.

We propose a framework for selecting and qualifying test models for the validation of model transformations [59]. We propose "black-box" test adequacy criteria for

this selection framework in order to leverage the complete description of the input domain provided by the source metamodel of the transformation. It is important that the proposed approach is generic and compatible with any model transformation language because there are many languages for transformation and none of them has emerged as the best or the most popular. The proposed criteria can be used to qualify test data for model transformations implemented with a general purpose language such as Java, the specific model transformation language QVT [112] proposed by the OMG, a metamodelling language such as Kermeta [102], a rule-based language such as Tefkat [50], or a graph transformation language such as ATOM3 [40]. The second reason why we choose black box criteria is to leverage the fact that the input domain for a transformation is defined by a metamodel. Indeed, the source metamodel of a transformation completely specifies the set of possible input models for a transformation. In this context, the idea is to evaluate the adequacy of test models with respect to their coverage of the input metamodel. For instance, test models should instantiate each class and each relation of the input metamodel at least once.

Models are complex graphs of objects. To select useful models we first have to determine relevant values for the properties of objects (attributes and multiplicities) and next to identify useful structures of objects. For the qualification of values of properties we propose to adapt a classical testing technique called category-partition [113] testing. The idea is to decompose an input domain into a finite number of sub-domains and to choose a test datum from each of these sub-domains. For the definition of object structures, we propose several criteria to assemble properties and form pieces of model that should be covered by the test models.

An important contribution of this work consists in defining a metamodel that formally captures all the important notions necessary for the evaluation of test models (partitions and object structures). This metamodel, given figure 4.4, provides a convenient formal environment to experiment different strategies for test selection, and a framework that checks if test models are adequate for testing. It distinguishes two types of partitions modelled by the classes VALUEPARTITION and MULTIPLICITYPARTITION that respectively correspond to partitions for the value and the multiplicity of a property. For a MULTIPLICITYPARTITION, each range is an integer range (class INTEGERRANGE). For a VALUEPARTITION, the type of ranges depends on the type of the property: STRINGRANGE, BOOLEANRANGE, INTEGERRANGE).

The metamodel defines the notions of model fragments (MODELFRAGMENT), object fragments (OBJECTFRAGMENT) and property constraints (PROPERTYCONSTRAINT) to represent combinations of partition ranges. A model fragment is composed of a set of object fragments. An object fragment is composed of a set of property constraints, which specify the ranges from which the values of the properties of the object should be taken from. It is important to note that an object fragment does not necessarily define constraints for all the properties of a class, but can partially constrain the properties (like a template).

On the basis of this metamodel, we have developed the metamodel coverage checker (MMCC) that automatically analyses a set of test models and provides the testers with

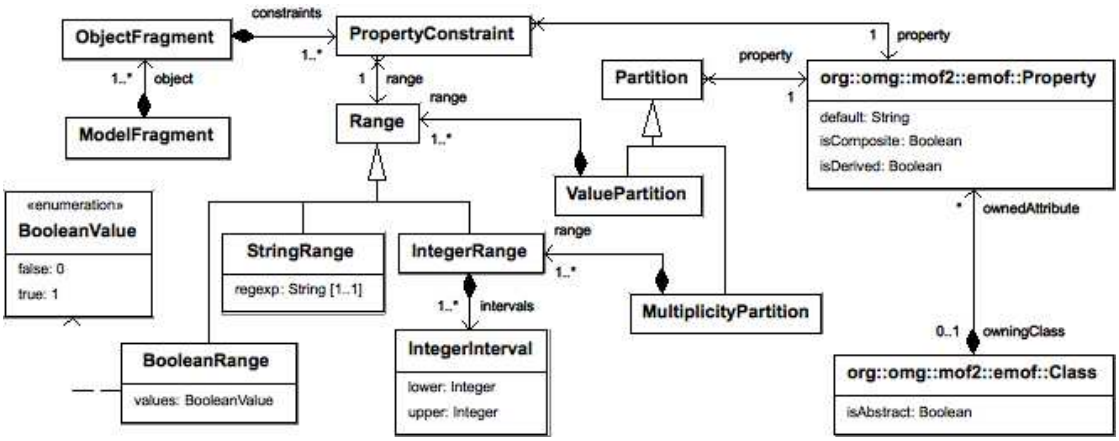

Figure 4.4: Metamodel for the definition of coverage criteria for test models

valuable feedback concerning missing information in their test models. This information can then be used to iteratively complete a set of test models. It takes two inputs: the source metamodel of the transformation under test and a set of test models. From the input metamodel, MMCC generates the default partitions for all features contained in the metamodel, then combines these partitions to build a set of model fragments. During both partitioning and combination the tester may enrich the generated models to take domain specificities of the transformation under test into account.

When the model fragments are generated from the input metamodel, MMCC checks that there is at least one test model that covers each model fragment. If there are fragments not covered by the test models, the tester should improve the set of test models by adding new models to cover the identified remaining fragments. This process does not only allow for an estimate of the quality of a set of test models but also provides the testers with valuable information to improve the test set.

4.2.2 Automatic test data generation

We explore greedy [37] and constraint-based approaches [126] for automatic generation of models that cover all model fragments. The greedy algorithm generates all fragments for a source metamodel and then assembles them in order to form complete models for testing a model transformation [37]. It is necessary to fix the number of fragments in the generated models: between one large model that covers all object fragments and a set of small models, each one covering one model fragment. After each object fragment is covered, it is necessary to complete the model. Here we have adapted Tarjan's algorithm [138] for avoiding cycle issues when adding objects to complete the model.

This initial experiment provide valuable insights on the difficulties for automatic generation of test models. However, the greedy approach has a number of limitations, in particular the identification of fragments that can be combined together. Also, the

generated models conform to the metamodel but do not satisfy the additional OCL constraints that can be specified on the metamodel. These observations lead us to investigate constraint-based techniques for this automatic generation task [126, 127].

For this second exploration of automatic model synthesis the major issue is due to the large amount of constraints that these models have to satisfy. There are two kinds of constraints that must be considered for test models: the constraints that define the licit input models for the transformation and the constraints that aim at selecting models with a specific testing goal. In addition to this large set of constraints, another challenge consists in dealing with the heterogeneous formalisms in which these constraints are expressed.

The solution studied in this work focuses on four types of constraints expressed in different formalisms, as illustrated in figure 4.5. Two types of constraints define the set of licit models for the transformation: the metamodel and the pre condition. The metamodel is specified in two parts: a structure built with the ECORE language [52], and constraints on this structure expressed in OCL. The pre-condition for the transformation is also expressed in OCL. Two types of constraints are used to select test models among the whole set of licit models: partitions on the input domain and test model objectives that are derived from the requirements of the transformation. The partitions are derived from the metamodel and are composed in model fragments according to the test criteria defined in [59]. Currently, there exists no particular modelling language to specify the requirements for a transformation or to express test model knowledge. Thus, we model them directly in ALLOY [73], which is the underlying language for test model synthesis.

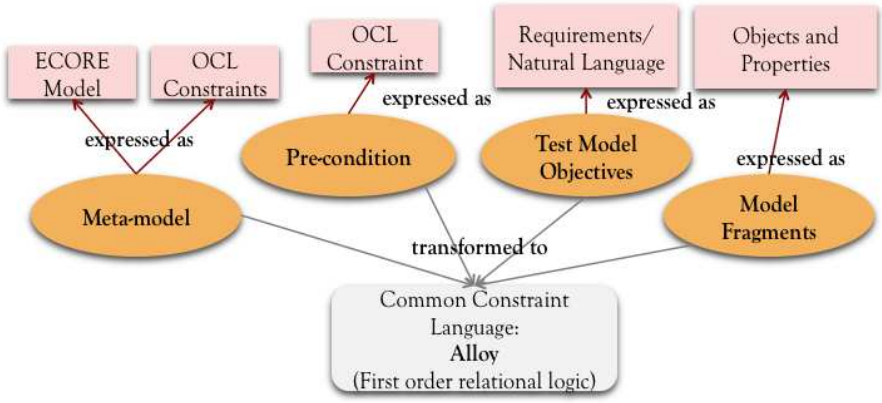

Figure 4.5: Combining heterogeneous constraints for automatic model synthesis

This approach allows the automatic generation of models that cover all the model fragments and that satisfy all constraints that specify the input domain of the transformation. Even if it has the usual scalability issue of constraint-based approaches, it is much more effective than the previous approach because all the generated models can effectively be passed to the transformation as input test data. We develop the Pramana tool [124] that bridges the gap between the ECORE and ALLOY worlds for automatic

model synthesis.

4.2.3 Oracle for model transformation testing

The oracle checks the validity of the output model returned by the transformation of one test model. It analyzes a model and returns the verdict for the test case.

Few works mention the oracle function for model transformation, and they usually consider that the expected model for a particular run of the transformation is available [71, 89]. Thus, they transform the problem of oracle definition into a problem of model comparison. Although this approach has to be considered and efficient solutions for model comparison will help the definition of an oracle function, we believe that considering the oracle only through this perspective is too restrictive. First, the expected model is not easy to obtain, and the tester might face difficulties to produce expected models for all the test cases. In practice this can hardly be used beyond regression testing (in that case the expected model can be produced by the previous version of the transformation). Second, there are several other ways to analyze the output model and produce a verdict that should not be neglected because they could fit more easily the tester needs. Finally, we believe that a model transformation testing oracle should not be reduced to a data but has to be considered as a full function with its parameters.

In order to analyze the definition of an oracle in a broader way than simple model comparison, we consider the oracle as a parameterized function [99]. The first parameter is the output model returned by the transformation. The second parameter must be provided by the tester, and we call it the "*oracle data*". This data provides details to verify the output model. It is the main parameter of the oracle. For instance, it can be the expected model of the test case; it can also be the test model if an oracle needs to extract information from it to check the output model.

In the following, we analyze the different data that can be provided and the different functions that can be defined, depending on the oracle data.

Three MDE techniques to implement oracle functions

We introduce three Model Driven Engineering (MDE) techniques that manipulate and analyze models and that can be used to implement model transformation oracles.

Model comparison Alanen et al. [4] present a theoretical framework for performing model differencing relying on the use of unique element identifiers for the model elements. Other algorithms based on the metamodels despite the objects identifiers have been proposed. In [89], Lin et al. proposed such an algorithm. Model comparison is implemented in tools like EMFCompare [62].

Contracts Several researchers have studied the use of contracts as a partial oracle functions in object oriented system [33, 86]. We propose a process for specifying and implementing model transformations oracle with contracts expressed in OCL [98]. Kolovos et

al. [81] link the output and the input models with rules. Küster et al. [83] have also noticed that constraints can be used as oracle.

Pattern matching We consider patterns expressed as OCL assertions or as model snippets [119, 99]. For the oracle, the patterns express constraints on the output model. They can be considered as assertions that should be true when running the transformation with a particular test data. Each assertion or a conjunction of several ones can be associated to a test case as the oracle data of an oracle function.

Six oracle functions for model transformation

Six solutions are thus available to obtain the oracle when executing a test data on a model transformation.

1. Oracle using a *reference model transformation*:

   Compares the output (`mtout`) of the transformation with a reference model (`mt`) generated by a reference model transformation (`R`). The oracle function O_1 is defined as:

```
O1(mtout , R, mt): Boolean is
do
    result := compare(mtout , R(mt))
end
```

2. Oracle using an *inverse transformation*

   Compares the test model `mt` and the model generated by two successive: the first with the transformation under test and the second with the inverse transformation (`I`). The oracle function O_2 is defined as:

```
O2(mtout, I, mt): Boolean is
do
    result := compare(mt , I(mtout))
end
```

3. Oracle using an *expected output model*

   Compares the output model (`mtout`) with an *expected model* (`mtexpected`) provided by the tester. The oracle function O_3 is defined as:

```
O3(mtout, mtexpected): Boolean is
do
    result := compare(mtout, mtexpected)
end
```

4. Oracle using a *generic contract*

   A generic contract `Cg` is a post condition of the transformation which constrains the outputs depending on the inputs. `Cg` checks that the output model `mtout` is correct with respect to the test model `mt`. The oracle function O_4 is defined as:

```
O4(mtout, Cg, mt): Boolean is
do
    result := (mtout,mt).satisfies(Cg)
end
```

5. Oracle using an *OCL assertion*

   The oracle checks if the output model satisfies the OCL assertion (`Cd`). The oracle function O_5 is defined as:

```
O5(mtout,Cd): Boolean is
do
    result := mtout.satisfies(Cd))
end
```

6. Oracle using *model snippets*

   The oracle checks if the output model (`mtout`) contains `n` model snippets (`ms`). The oracle function O_6 is defined as:

```
O6(mtout,list\{(ms,n,op)}): Boolean is
do
    //compares 2 numbers depending on a logical operator op
    //returns a boolean
    result := list.forAll(compare(nb\_match (mtout,ms), n, op))
end
```

These 6 functions can be instantiated for various model transformation languages. The choice of or another function depends mostly on what information is available (inverse transformation, contracts) and the necessity for evolution.

4.3 Feedback to model transformation engineering

We faced a number of lacks in MDE tools when developing our testing contributions. In some cases we tackled these gaps with original contributions to MDE, and used them to develop model transformation solutions. We develop two techniques related to the automatic generation of test data. The first one is related to the observation that most transformations declare a source metamodel which is actually much larger than the actual, effective source metamodel. This is a major issue for automatic generation of source models since there is a great chance that the generator produces models which do not make any sense for the transformation under test. Thus, we specify and implement

a metamodel pruning algorithm [130] that can 'extract' the effective source metamodel from a large source metamodel. Second, we often feel the need to specify pieces of models that we would like to have in a test model, but we do not really want to specify the rest of the model because it should not impact the test case. In order for a tester to do that we propose a tool for automatic completion of model [129].

The third feedback we provide to MDE wraps up all our proposals for testing in a general model that encapsulate tests, contracts and transformation in a trustable component [98].

4.3.1 Metamodel pruning

We present a *metamodel pruning algorithm* [130] that takes as input a large metamodel and a set of required classes and properties, to generate a target *effective metamodel* [61]. The effective metamodel contains the required set of classes and properties for a model transformation to proceed correctly. The term *pruning* refers to removal of unnecessary classes and properties. From a graph-theoretic point of view, given a large input graph (large input metamodel) the algorithm removes or prunes unnecessary nodes (classes and properties) to produce a smaller graph (effective metamodel). The algorithm determines if a class or property is unnecessary based on a set of rules and options. One such rule is removal of properties with lower bound multiplicity 0 and who's type is not a required type.

Given a set of required classes and properties the rationale for designing the algorithm was to remove a maximum number of classes and properties facilitating us to scale a formal method to solve constraints from a relatively small input metamodel. For instance, we remove all properties which have a multiplicity 0..* and with a type not in the set of required class types. However, we also add some flexibility to the pruning algorithm. We provide options such as those that preserve properties (and their class type) in a required class even if they have a multiplicity 0..*. Whatever option is chosen, the resulting metamodel is a supertype of the large input metamodel [133], and identical meta-concept names are preserved. These properties ensure backward compatibility of the effective metamodel with respect to the large input metamodel.

Figure 4.6, displays an overview of the pruning algorithm. The inputs to the algorithm are: (1) A source metamodel $MM_s = MM_{large}$, (2) A set of required classes C_{req} (3) A set of required properties P_{req}, and (4) parameters to make the algorithm flexible for different pruning options

The set of required classes C_{req} and properties P_{req} can be obtained from various sources: (a) A static analysis of a model transformation can reveal which classes and properties are used by a transformation (b) The sets can be directly specified by the user (c) A model itself uses objects of different classes. These classes and their properties can be the sources for C_{req} and P_{req}.

The output of the algorithm is a pruned effective metamodel MM_t=$MM_{effective}$ that contains all classes in C_{req}, all properties in P_{req} and their associated dependencies. Some of the dependencies are mandatory such as all super classes of a class and some

are optional such as properties with multiplicity 0..* and whose class type is not in C_{req}. A set of parameters allows us to control the inclusion of these optional properties or classes in order to give various effective metamodels for different applications. The output metamodel $MM_{effective}$ is a subset and a super-type of MM_s.

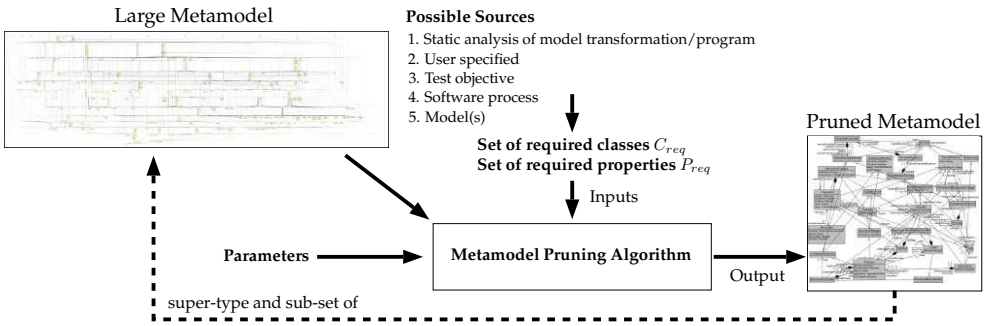

Figure 4.6: A metamodel pruning process

We use the notion of model typing that to demonstrate that the generated effective metamodel, a subset of the large metamodel from a set-theoretic point of view, is a *super-type* , from a type theoretic point of view, of the large input metamodel. This means that all programs written using the effective metamodel can also be executed for models of the original large metamodel. The extracted effective metamodel is very much like a transient DSML with necessary concepts for a problem domain at a given time.

	Original UML	No option	Option 1	Option 2	Option 3
Number of classes	246	31	31	31	31
Number of properties	583	15	26	30	30

Table 4.1: Pruning the UML for a CNES transformation

As an example, we present an application of our algorithm to generate an effective metamodel to specify test models for a model transformation. The model transformation is developed by the French Spatial Agency (CNES) to transform UML models to code for embedded systems. Table 4.1 shows some metrics on the number of classes and properties when pruning the UML metamodel for this transformation. Options 1, 2 and 3 are options on properties that are selected or not, depending on their cardinality. For example, a property that has a lower bound 0 can be selected or not.

4.3.2 Automatic model completion

Generative modelling tools such as AToM3 (A Tool for Multi-formalism Metamodelling) [40], GME(Generic Modelling Environment) [88], GMF (Eclipse Graphical modelling

Framework) [53] can synthesize a domain specific visual model editor from a declarative specification of a domain specific modelling language. A declarative specification consists of a metamodel and a visual/textual syntax that describes how language elements (objects and relationships) are represented in the model editor. The designer of a model uses this model editor to construct a model on a drawing canvas. This is analogous to using an integrated development environment (IDE) to develop a program or a word processor to enter sentences. However, IDEs such as Eclipse present recommendations for completing a program statement when possible based on its grammar and existing libraries [80]. Similarly, Microsoft Word presents grammatical correction recommendations if a sentence does not conform to natural language grammar. Here we investigate the possibility of proposing recommendations for model completion.

The major difficulty for providing completion capabilities in model editors is to integrate heterogeneous sources of knowledge to synthesize correct recommendations. The completion algorithm must take into account the concepts defined in the metamodel, constraints on the concepts in the metamodel and the partial model built by a domain expert/user. These three sources of knowledge are obviously related (they refer to the same concepts) but are expressed in different languages, sometimes in different files, and in most cases by different people and at different moments in the development cycle as they are separable concerns. We propose an automatic transformation from all these sources of knowledge to an Alloy [73] model. The generated Alloy model is then used to synthesize a set of Boolean CNF formulae by the KodKod engine [140] available in the Alloy Java API. Solving this set of Boolean CNF using a satisfiability (SAT) solver returns one or more possible solutions for completing the model.

Our transformation from the different sources to Alloy is integrated in the software tool AToM3. The metamodel for a DSML is built directly in AToM3's model editor using its class diagram formalism. The constraints on the concepts of this metamodel are defined using Alloy facts. Using this information and a description of the concrete visual syntax (specified in an icon editor) for a modelling language, AToM3 synthesizes a visual model editor for the DSML. The partial model can be built and edited in the generated model editor and the designer can ask for recommendations for possible automatic completions [129].

Figure 4.7 displays an example. A modeller or a tester specifies she wants a model, which has two states, related by a transition, two other states related by another transition and an additional state. Our tool transforms this piece of a model to a set constraints and encodes all information in the metamodel in a set of additional constraints. Then, it solves the whole set of constraints in order to generate a complete model that includes the initial piece of model as well as elements that make it a complete model correct with respect to the metamodel.

4.3.3 Encapsulating model transformations into components

Based on the analysis and testing contributions detailed above, we propose a model to encapsulate model transformations in components called MDA components. The contri-

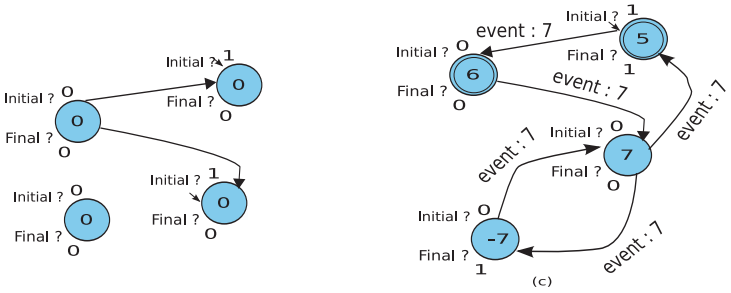

Figure 4.7: Initial and completed models

bution of this work is to propose a model for trustable components as well as a methodology to design and implement such components [98]. Trustworthiness is a general notion, which includes security, testability, maintainability and many other concerns. Here, we claim that trust is, in first, dependent on the quality of the tests in relation with the completeness of the specification, captured by executable contracts (as defined in design-by-contract [92]). While building MDA components, we consider it as an organic set composed of three facets: test cases, an implementation and contracts defining its specification. A trustable component is considered as being "vigilant", in the sense it embeds contracts efficient enough to detect most of the erroneous states at runtime. Trust is evaluated using testing-for-trust process and reflects the consistency between the specification and the implementation of the component.

The model for trustable MDA components is based on an integrated design and test approach for software components. It is particularly adapted to a design-by-contract [92] approach, where the specification is systematically translated as executable contracts (invariant properties, pre/postconditions of methods). In this approach, test cases are defined as being an "organic" part of a component: a component is composed of its specification (documentation, methods signature, invariant properties, pre/postconditions), one implementation and the test cases needed for testing it. Figure 4.8 illustrates this view of a component with a triangle representation.

From a methodological point of view, we argue that the trust we have in a component depends on the consistency between its three facets. The confrontation between these three facets leads to the improvement of each one. First, a good set of test cases is generated. Then it is possible to improve the implementation: running the good test cases allows to detect faults and to fix them. At last the accuracy of contracts can be improved to make them effective as an oracle function for test cases.

The trust in the component is thus related to the test data effectiveness and the contracts "completeness". We can trust the implementation since we have tested it with a good test data set, and we trust the specification because it is accurate enough to derive effective contracts as oracle functions.

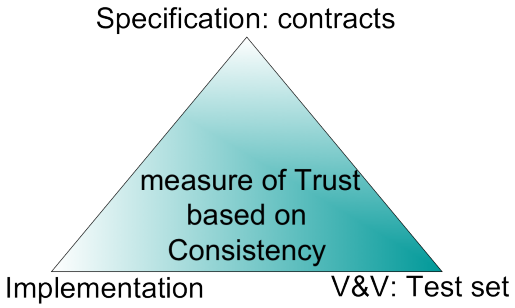

Figure 4.8: rust based on triangle consistency

4.4 Conclusion

Our work around testing and model transformation has expanded over 6 years (2004 [61] - 2010 [131]) and is the most balanced representation of the QLTF pattern. It started at a time where we could hardly develop automatic model transformations, and certainly not test them, while there are currently a large number of model transformation tools and there are more work on testing. Thinking about the question of testing at the emergence of model transformation techniques in MDE drove the development of the Kermeta language that integrated verification concerns right from its initial design in Franck Fleurey's PhD [57]. This platform allowed us to experiment innovative ideas both in the field of testing with the construction of test coverage tool for transformations [59] and in the field of transformations with the definition of a metamodel pruning algorithm [130] and the introduction of footprints for model operations [74]. All these work also gave us enough understanding of both fields to precisely characterize the remaining challenges for model transformation testing and propose future research directions [16]. In section 5.2 I detail some of these directions that I will explore in future work.

Chapter 5

Conclusion and Perspectives

5.1 Conclusion

There have been, there are and there always will be new paradigms to build software intensive systems. These paradigms emerge to fit new domains, to manage the complexity of requirements or to suit different types of developers. They aim at improving quality, increasing reuse or enhancing communication. However, they usually cannot prevent the presence of errors in software and, it is necessary to develop verification techniques along with the definition of new paradigms. One could argue that verification techniques are generic and should not need specialization for each paradigm. Our claim is that defining specialized verification techniques allows to look for the specific faults introduced by each paradigm; this allows providing tools that developers can understand and use if they are building software in one particular paradigm; we also believe that thinking about verification right from the definition of a paradigm allows one to improve the quality of software developed in this paradigm.

The work presented here focuses on verification through testing. It summarizes the investigations of testing strategies for object-oriented programming (OOP), aspect-oriented programming (AOP) and model transformations. To reflect the tight relationship between a particular paradigm for software construction and the associated testing techniques, we introduce the *Question-Learn-Test-Feedback* pattern (QLTF). This pattern captures the idea that, to test within paradigm, it is necessary to understand how it works, what is it meant for and what are the constraints for testing. For each paradigm, our work to develop testing techniques starts with a set of questions. These questions drive our initial experiments within a paradigm and lead to learning the characteristics of this paradigm that will be useful for testing. What we learn can lead to (1) the proposal of testing techniques that target specific errors in the paradigm; (2) feedback to the paradigm in the form of new constructs that make it more testable.

The QLTF pattern led to the following contributions in three paradigms

Object-oriented programming

- *Learn*. We learned about testability of object-oriented design models in the presence of Design Patterns and we analyzed the impact of Design by Contract on the quality of object-oriented programs [17, 86].

- *Test*. This understanding of the object-oriented paradigm led to the definition of the following test techniques: a bacteriologic algorithm for the automatic improvement of unit test suites [19, 12, 11, 13, 15]; a new test criterion to drive the generation of test cases that can optimize the accuracy of fault localization algorithms [87, 60, 14]; tools to support experiments with these proposals.

- *Feedback*. We also provided feedback to object-oriented design through the proposals of a UML profile to improve early testability [21, 20, 22, 18].

Aspect-oriented programming

- *Learn*. We performed an empirical analysis of open source aspect oriented programs to learn about the usage of AOP [108].

- *Test*. This allowed us to better understand how to test these programs and led to three proposals for testing: a static analysis to understand the impact of aspect weaving on test suites [46]; a specific oracle for testing pointcut descriptors in AspectJ; we also built a mutation tool to evaluate the fault detection abilities of our proposals [42].

- *Feedback*. We also proposed feedback to the aspect-oriented paradigm in the form of a framework for the specification of interactions between aspects and base program [105, 107]; we specified a measurement environment for aspect-oriented programs.

Model transformation

- *Learn*. We learned about the barriers for testing model transformations, which mainly lie in the complex nature of input / output data, the heterogeneity of transformation languages and the lack of support in model management environment [16]. In order to drive our investigation of testing techniques we also learned about the particular fault models in transformations[97].

- *Test*. We proposed three contributions to testing transformations: black-box coverage criteria over the input domain of a transformation [61, 59]; an imperative algorithm for the automatic generation of test models [37], and SAT-based technique [126, 127]; a family of techniques for the oracle [99].

- *Feedback*. While developing our work on testing, we made proposals for feedback to the model transformation community: a metamodel pruning algorithm for the

identification of a precise input domain [130]; a tool for automatic completion of models [128, 129]; a component model for transformations [98].

As a concluding remark about all these application of the QLTF pattern, we can notice that some elements occurred in all cases, which might reveal general knowledge about the integration of software testing in various design contexts. The first element that was always present as a key enabler for the integration of design and test is the presence of contracts. For object-oriented programming, aspect-oriented programming and model transformations, contracts were always useful to improve the design as well as to help testing with more information about the input domain and the expected result. The other element that was useful in all our studies was the systematic investigation of dedicated fault models. This helped both to understand what is the objective for testing and to evaluate the quality of the proposed testing solutions.

Two major trends in emerging software construction paradigms consist in (i) developing effective techniques that facilitate the expression of separate concerns with heterogeneous formalisms while (ii) increasing the degree of variability among all concerns. These two trends address the new challenges imposed by software intensive systems that are heterogeneous and prone to change due to their adaptive nature.

In future work I will develop a balanced activity between extensions of MDE and innovative verification techniques to address the construction of software-intensive systems. Next section provides details about these future work.

5.2 Perspectives

Major foundational works have set Model-Driven Engineering (MDE) as a sound approach for designing, verifying and building software systems [54, 6, 134, 36]. Current challenges for MDE emerge from the characteristics of future software-intensive systems. These systems have tight interactions with their environment, are heterogeneous by nature (they capture interactions between humans, mobile devices, physical environment, etc.) and prone to frequent changes (devices come in and out, actors change the requirements, the environment conditions change) [146]. In order to address the construction and the analysis of these systems, MDE has to develop new mechanisms to improve separation of concerns with heterogeneous formalisms and to allow high degrees of variability in models.

These new requirements for MDE and software-intensive systems (heterogeneous models and high degree of variability) will have a major impact on V&V:

- *separation of concerns*. On one hand, the generalisation of separation of concerns in models encourages the construction of several views on the model, expressed in different formalisms. On the other hand, existing V&V techniques such as model checking or testing assume one complete, homogeneous behavioural model. This gap is a serious issue for the verification of complex systems. Recent work by Krahn et al. [82] propose new approaches for the composing of languages. Model-based

testing is also an example of language composition for verification: it can compose a state-based model, structural constraints and a class diagram to generate test cases [142]. These approaches have to be generalised in order to allow the integration of new formalisms for V&V and thus adapt V&V to the new requirements of future systems.

- *high degree of variability*. The increased degree of variability at different levels of abstraction introduces a high degree of uncertainty and even inconsistency in models [28]. This prevents using current solutions for V&V which require the definition of a consistent, stable model. Recent studies investigate the verification of highly variable systems. For example, in the very dynamic context of web services Rosario et al. [121] introduce 'soft' contracts that can specify expected values over unstable quality of service, or Kattepur et al. [77] propose a systematic sampling of the QoS variability space to estimate global quality. These efforts must be extended in order to address verification and variability in an integrated, systematic way.

In future work I will investigate verification in heterogeneous, highly variable models, following the QLTF pattern (figure 5.1). The discussions above emphasize the gap between the current state of V&V and the direction in which MDE is going in order to address the challenges of emerging software-intensive systems. In other words, we know in which direction these systems want to go, we are currently starting to build the abstractions and MDE mechanisms that will assist their development, but we cannot know today how these systems will be verified. A consequence of this gap is that the *Learn* and *Feedback* facets of the QLTF pattern will be at the core of my research in the short term. This will aim at setting the techniques on which I will investigate innovative V&V techniques for heterogenous, variable models in the mid term.

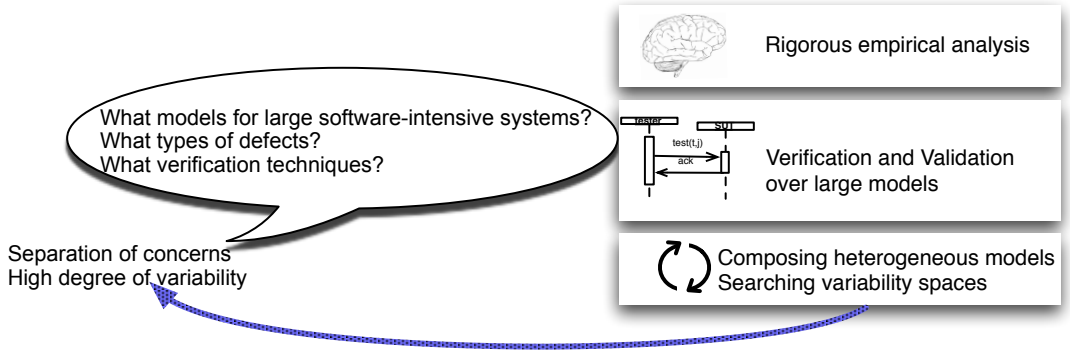

Figure 5.1: QLTF for future MDE in software-intensive systems

The next 4 sections provide detailed perspectives for MDE mechanisms that aim at addressing the challenges of software intensive systems while setting the fundamental

infrastructure for V&V. They are synthesized in figure 5.2. Section 5.2.1 proposes future work for the composition of heterogeneous models, and section 5.2.3 summarizes the directions I will follow to deal with large variability spaces in models. Sections 5.2.2 and 5.2.4 introduce two orthogonal perspectives: section 5.2.2 emphasizes the need to introduce human knowledge in order to build large and sustainable models; section 5.2.4 insists on rigorous experimentation as the scientific foundation for this project. Section 5.2.5 opens broader perspectives for future research on V&V, inspired by a series of recent work that draw a strong analogy between software engineering and biological phenomena.

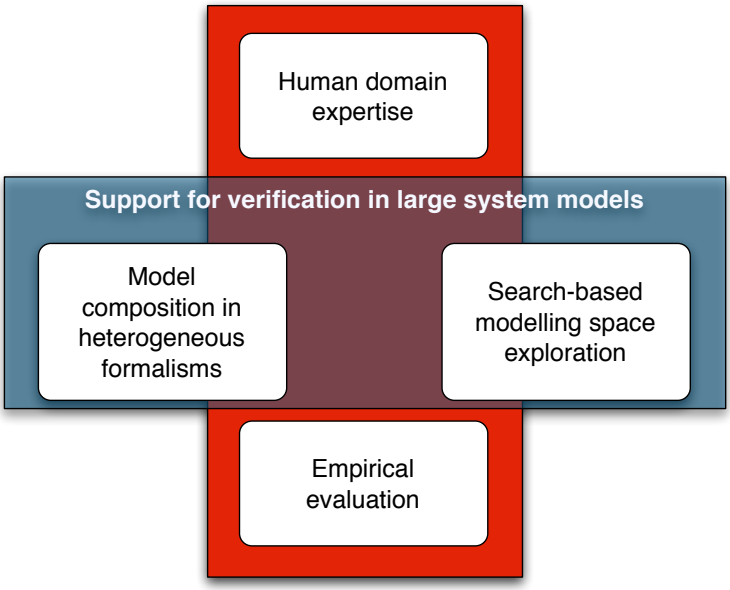

Figure 5.2: Investigating foundations for verification and analysis in large models

5.2.1 Model composition over heterogeneous domains

Separation of concerns and domain-specific modelling are necessary to handle the complexity of large-scale models. However, they imply a major consequence: when analyzing a global model that aims at capturing several concerns, it is necessary to understand how each concern relates to the others, and thus to understand the semantics of the different domain-specific modelling languages (DSMLs) with respect to the others. This means understanding the relationships between models and their respective metamodels with associated semantics.

In future work I will investigate the formalization of abstract composition operators to understand the relationships between models and metamodels. This will include the following points for investigation

- Understand the foundations of model composition. Currently there exist a large number of model composition approaches, but all of them are dedicated to one modelling formalism (*e.g.*, class models or statecharts) and one intention for composition (*e.g.*, fusion or weaving). The goal here is to understand what unifies all these composition mechanisms and to define a generic framework for model composition.

- Model typing to deal with heterogeneous formalisms. In order to compose a pair of models that are specified with two different DSMLs, it is necessary to understand the relationship between the languages. In this context, the composition of metamodels can be viewed as a model type, that synthesizes some concerns common to these DSMLs. This type can serve for the definition of model manipulations that are reusable on the different DSMLs. The goal here would be to extend the work on model typing to allow the definition of semantic links between types and to allow the sound composition of DSMLs

- Relationships between metamodels and models. Assuming it is possible to compose types on one hand and to compose models that conform to these types on the other hand, the first composition operation sets constraints over the second one. This means that the composition of a pair of models depends on the relationships that have been established between the pair of DSMLs. The goal here is thus to set operators that can interpret the relations between types and use them to constrain or drive the composition of models that conform to these types.

5.2.2 Bring domain expertise in model manipulation

When implementing an operation that automatically manipulates models (refinement, translation, checking, etc.) one always has to face a complex trade-off between genericity and domain specificity. Genericity offers the advantage of a manipulation that can run on a variety of models (*e.g.*, in different projects or in different formalisms). On the other hand, considering domain specific elements for the manipulation can allow the computation of a more relevant result. In a context where the number of application domains and modelling contexts keeps growing, it is very difficult to find the best trade-off. An alternative can consist in building manipulations which precisely identify points that can be tuned to a particular context. When the manipulation executes, a human domain expert, provides the information.

This future work will investigate three different ways of bringing domain expertise in model manipulation: specific modelling concepts, dedicated languages, interactive modelling. These three approaches are detailed below.

- We have recently investigated the explicit modelling of intention in modelling activities [103]. This work focuses on the idea that “the purpose of a model must be understood before the model can be discussed” [122]. This means that human modellers always have an intention in mind when they build a model, and this intention can be more important for understanding than the information that is put

in the model. Our work introduces a formal model of different intentions that can exist between models. In the future we would like to experiment how this explicit knowledge of intention can be used. For example, modellers can add an intentional dimension in models in order to understand what changes are acceptable according to the initial intention, in case the system has to evolve.

- A practical way to let domain experts bring their knowledge in model manipulations is to offer them languages that are close to their domains and that can be automatically translated in the terms of the manipulation. These languages can take the form of a subset of English, with pre-defined template sentences. The set of authorized sentences is specific to one domain, and each sentence can be associated to a set of rules to translate the sentence in a formal expression usable by the manipulation. A specific case of such template sentences is known as boilerplates [72] that are used to express requirements in a form that can be automatically simulated and checked for logical inconsistencies.

- Domain experts possess valuable information about what they expect as a good solution for a model manipulation. However, this knowledge is seldom formalized in a way that manipulations could use it automatically. A possible approach to still benefit from this knowledge when running the manipulation is to have an interactive approach in which the tool proposes solutions, asks the expert's opinion and take it into account to propose better solutions. This approach is tightly related to search-based approaches and we have started investigating such interactions in the context of invariant discovery for the specification of domain models (Juan Cadavid's PhD started in November 2009)

5.2.3 Search-based exploration of variability in modelling spaces

The more concerns we will be able to compose in models, the more views we will be able to integrate, the more complex it will be for modellers to understand all interactions and all impacts these concerns have on functional or qualitative properties of the global model. In addition to these concerns, the presence of variability in models increases the number of interactions and thus the size of the global space for verification. For example, this growth occurs in models for software product lines, service-oriented architectures and self-adaptive systems. Human-based reasoning cannot deal with this growing number of interactions, interferences and system variants. This future work will consist in investigating constraint programming and meta heuristics to explore modelling spaces and efficiently search for good solutions in the context of verification tasks.

- One important task for this future work is to understand which distance measures can exist between models. This will serve, for example, to evaluate the similarity between models during the automatic exploration of a modelling space. In this task we will analyze how distances between models can be adapted from existing similarity measures between graphs. In particular we will investigate how the fact

that models are typed attributed graphs (which types relations are specified by a metamodel) can help in taming the general complexity of graph distance.

- Constraint programming offers powerful theories and tools to specify customized strategies for the exploration of large modelling spaces. For example, we have recently investigated a SAT-based sampling of the variability space. We have demonstrated the feasibility of the approach [116] and a possible application for QoS evaluation in composite web services [77]. We plan to continue investigating constraint-based techniques to verify highly configurable components in the context of AUTOSAR. These work open a more general investigation about the introduction of constraint programming for model-driven engineering and the definition of a constraint language for automatically reasoning about models.

- Meta heuristics are useful to address complex optimization problems such as finding a good trade-off between different concerns that should be integrated in a system model. They also propose an alternative way to explore large modelling spaces. For example, we experimented with meta-heuristics in Freddy Muñoz's PhD thesis: he proposes new criteria to sample the environment of self-adaptive systems for testing and he defined three strategies based on genetic and bacteriologic algorithms to automatically generate the samples [104].

5.2.4 Rigorous empirical validation

This point in future work is more 'meta' and is here to remind that these research topics are deeply rooted in Software Engineering. This means in particular that all the foundations we set, the methods we define, the techniques we develop must be validated through rigorous experimental methods and with respect to their initial goal: provide support for verification and analysis of large system models. Scientific methods have been precisely established in the fields of natural sciences. They are based on the assumption that the construction of reliable knowledge in a field has to be supported by a series of observations, a strict control over independent variables. Similar investigation methods must be applied to software engineering in order to build a sound body of knowledge and eventually extract theories for large-scale model-driven engineering.

Experimental investigations can serve two goals:

- A series of precise experiments can help us understand the current state of practice in MDE. This should allow us to qualify and quantify what can currently be done with MDE and thus, what are the initial assumptions for our work. This initial purpose of experiments is directly related to the *Question* and *Learn* dimensions of the QLTF pattern.

- In future work we will propose new contributions to MDE in the form of algorithms, tools or methods. Since all they all aim at improving the adoption of MDE for the verification and analysis of large system models, they have to be validate

with respect to that goal. The second purpose of experiments is thus to provide evidence that the work we develop can serve its initial goal.

5.2.5 Validation and verification of software intensive systems

As mentioned at the beginning of this section, the major challenge for V&V resides in the fact that current techniques assume a single, complete, homogeneous model. In previous sections I have introduced perspectives on model composition in order to extract a global view from heterogeneous perspectives and in section 5.2.3 I have introduced possible leads to deal with variability dimension in these models. However, if we can globally manipulate heterogeneous perspectives on a system, we still do not know what it means to verify its behaviour and how to perform this verification.

In order to verify software intensive systems I will investigate possible analogies between the 'correctness' of biological systems and the verification in software intensive systems. This proposal comes from the intuition that the dynamic, adaptive, pervasive nature of software intensive systems can be related to living phenomena. Thus, understanding the mechanisms that biological systems develop in order to detect intruders and adapt for survival can inspire innovative procedures to analyze and fix software intensive systems. This proposal is also motivated by a series of recent investigations for bio-inspired solutions to address the complexity of software systems. I briefly present these work in the following.

The analogy of software systems to various forms of biological phenomena has inspired computer scientists and software engineers for a long time. It has driven innovative ways of building and verifying programs (*e.g.* genetic programming [145], chemical programming [7] or artificial immunology [139]), and has inspired the development of powerful heuristics (*e.g.* genetic algorithms) that can address hard combinatorial problems. Some very recent works dealing with large software-intensive systems look for inspiration within different natural phenomena.

- The European CONNECT project [39] stresses that the constant emergence of new mobile devices prevents the emergence of a unique, stable middleware for communication between all devices. They investigate artificial learning techniques to dynamically and automatically discover communication protocols [30, 1], in order to tackle this issue.

- The american National Science Foundation has recently launched the BEACON [23] multi disciplinary research center that aims at developing bio-computation solutions for the development of complex systems. In particular, Professor Cheng's group studies how natural evolution can inspire the design of self-adaptive software systems [69, 90].

When recognizing that the scale (in time and space) and heterogeneity of software systems can be similar to natural phenomena, one can start developing truly innovative views on software. For example, Misailovic et al. [94] question the discrete nature of software and propose to think about some software systems as continuous phenomena.

Consequently, correctness is not a binary property anymore and it is possible to perform minor changes in the functionality that will not prevent the program from running. Misailovic et al. have experimented loop perforation as a possible way to alter a program's behaviour (by skipping some iterations in loop) to deliver increased performance in image and video processing software.

Recent work has even leveraged the omnipresence of human effort in software-intensive systems for solving problems that computers cannot complete. Von Ahn et al. [3] have been able to channel the effort of millions humans all over the world to assist computers in scanning old printed documents. They use the information provided by millions of humans who read CAPTCHAs everyday to access web sites. These distorted pieces of text are meant to distinguish humans from computers in order to prevent abusive accesses to web sites. The idea of Von Ahn et al. is that pieces of old manuscript documents that cannot be recognized by Optical Character Recognition are good candidates as CAPTCHAs. This constitutes an original case of *symbiosis* where computers and humans are closely associated to the benefits of both: humans are able to help computers perform a task that will in turn be useful to humans (who can store, search, transfer the scanned documents).

5.3 Concluding remarks

This project will participate in the advancement and adoption of model-driven engineering for large trustable software-intensive systems. However, there are risks in this project. First, each point in the project presents inherent risks: model composition has been a research topics for years; interactions between humans and computers is always a difficult topic; by nature search-based techniques are prone to failure. Second, we might not be able to identify the correct abstractions to reason about these systems. Third, the experimental nature of the project is, in my opinion, an essential component to propose effective solutions for large models, but it is also inherently risky because it aims at encompassing heterogeneous aspects of modelling (the metamodeller, the modeller, the users, the changes in requirements).

Despite these risks, I am confident that this project will succeed. First, because we have started exploring some parts of this project: model composition [58, 64, 115], extensions of model type [130, 131], and verification in large, variable environments [104, 48]. Second, I believe that valuable knowledge will come out of this project because of its intent. Following a QLTF pattern can force scientific rigour in all explorations and thus drive this work towards new knowledge about the construction of large models, the integration of heterogeneous concerns and the degree of variability and uncertainty that can be managed in models. In other words, whatever we do, if done with rigour and valid methods, will lead to relevant discoveries, fun explorations and valuable contributions to the field of MDE and verification for software intensive systems.

Bibliography

[1] Fides Aarts, Johan Blom, Therese Bohlin, Yu-Fang Chen, Falk Howar, Bengt Jonsson, Maik Merten, Ralf Nagel, Antonino Sabetta, Siavash Soleimanifard, Bernhard Steffen, Johan Uijen, Thomas Wilk, and Stephan Windmuller. Establishing basis for learning algorithms. Technical Report, 02 2010.

[2] Hira Agrawal, Joseph Horgan, Saul London, and Eric W. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE'95 (Int. Symposium on Software Reliability Engineering)*, pages 143 – 151, Toulouse, France, 1995.

[3] Luis Von Ahn, Benjamin Maurer, Colin Mcmillen, David Abraham, and Manuel Blum. recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465 – 1468, September 2008.

[4] Marcus Alanen and Ivan Porres. Difference and union of models. In *UML'03 (Unified Modeling Language)*, San Francisco, CA, USA, 2003.

[5] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[6] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36 – 41, sep. 2003.

[7] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Principles of chemical programming. *Electr. Notes Theor. Comput. Sci.*, 124(1):133–147, 2005.

[8] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, 1996.

[9] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[10] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to .net components. In *ASE'02 (Automated Software Engineering)*, pages 253 – 256, Edimburgh, Scotland, UK, 2002. IEEE Computer Society Press, Los Alamitos, CA, USA.

[11] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Computational intelligence for testing .net components. In *Microsoft Summer Research Workshop*, Cambrige, UK, 2002.

[12] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Genes and bacteria for automatic test cases optimization in the .net environment. In *ISSRE'02 (Int. Symposium on Software Reliability Engineering)*, pages 195 – 206, Annapolis, MD, USA, 2002. IEEE Computer Society Press, Los Alamitos, CA, USA.

[13] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability*, 15(1):73–96, 2005.

[14] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *ICSE'06 (Int. Conference in Software Engineering)*, pages 82 – 91, Shanghai, China, 2006.

[15] Benoit Baudry, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. An original approach for automatic test cases optimization: a bacteriologic algorithm. *IEEE Software*, 22(2):76–82, 2005.

[16] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2010.

[17] Benoit Baudry, Jean-Marc Jézéquel, and Yves Le Traon. Robustness and diagnosability of designed by contracts oo systems. In *Metrics'01 (Software Metrics Symposium)*, pages 272 – 283, London, UK, 2001. IEEE Computer Society Press, Los Alamitos, CA, USA.

[18] Benoit Baudry and Yves Le Traon. Measuring design testability of a uml class diagram. *Information and Software Technology*, 47(13):859–879, 2005.

[19] Benoit Baudry, Yves Le Traon, Vu Le Hanh, and Jean-Marc Jézéquel. Building trust into oo components using a genetic analogy. In *ISSRE'00 (Int. Symposium on Software Reliability Engineering)*, pages 4 – 14, San Jose, CA, USA, 2000. IEEE Computer Society Press, Los Alamitos, CA, USA.

[20] Benoit Baudry, Yves Le Traon, and Gerson Sunyé. Testability analysis of uml class diagram. In *Metrics'02 (Software Metrics Symposium)*, pages 54 – 63, Ottawa, Canada, 2002. IEEE Computer Society Press, Los Alamitos, CA, USA.

[21] Benoit Baudry, Yves Le Traon, Gerson Sunyé, and Jean-Marc Jézéquel. Towards a safe use of design patterns for oo software testability. In *ISSRE'01 (Int. Symposium on Software Reliability Engineering)*, pages 324 – 329, Hong-Kong, China, 2001. IEEE Computer Society Press, Los Alamitos, CA, USA.

[22] Benoit Baudry, Yves Le Traon, Gerson Sunyé, and Jean-Marc Jézéquel. Measuring and improving design patterns testability. In *Metrics'03 (Software Metrics Symposium)*, Sydney, Australia, 2003. IEEE Computer Society Press, Los Alamitos, CA, USA.

[23] BEACON. Bio/computational evolution in action consortium. `http://www.beacon.msu.edu/`, 2010.

[24] K. Beck and E. Gamma. Test-infected: Programmers love writing tests. *Java Report*, 3(7):37 – 50, 1998.

[25] Kent Beck. *Extreme programming explained*. Addison-Wesley, 1999.

[26] Kent Beck and E. Gamma. Junit, 2001.

[27] Boris Beizer. *Black-Box Testing*. Wiley, john wiley & sons edition, 1995.

[28] Nelly Bencomo, Jon Whittle, Peter Sawyer, Anthony Finkelstein, and Emmanuel Letier. Requirements reflection: requirements as runtime entities. In *ICSE (2)*, pages 199–202, 2010.

[29] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In Lionel Briand and A. Wolf, editors, *Future of Software Engineering*, pages 85 – 103. IEEE Computer Society, 2007.

[30] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic Synthesis of Behavior Protocols for Composable Web-Services. In Hans van Vliet and Valérie Issarny, editors, *The 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) – ESEC/FSE09*, pages 141–150, Amsterdam Europe, 08 2009. ACM.

[31] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Mtip workshop, 2005.

[32] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns and Tools*. Addison-Wesley, 1999.

[33] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object oriented code. *Software Practice and Experience*, 33(7), 2003.

[34] Lionel Briand, J.W. Daly, and J.K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91 – 121, 1999.

[35] Lionel Briand and Yvan Labiche. A uml-based approach to system testing. *Software and Systems Modeling*, 1(1):10 – 42, 2002.

[36] Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. *Software and System Modeling*, 1(1):10–42, 2002.

[37] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *ISSRE'06 (Int. Symposium on Software Reliability Engineering)*, pages 85 – 94, Raleigh, NC, USA, 2006.

[38] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, Verification Meyer, BertrandSoftware Testing, and n/a. doi: 10.1002/stvr.415 Reliability. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 2009.

[39] CONNECT. Emergent connectors for eternal software intensive networked systems, 2009.

[40] Juan de Lara and Hans Vangheluwe. Atom3: A tool for multi-formalism and metamodelling. In *FASE '02 (International Conference on Fundamental Approaches to Software Engineering)*, pages 174–188, 2002.

[41] Romain Delamare. *Analyses Automatiques pour le Test de Programmes Orientés Aspect*. Phd, 2009.

[42] Romain Delamare, Benoit Baudry, Sudipto Ghosh, Shashank Gupta, and Yves Le Traon. An approach for testing pointcut descriptors in aspectj. *Journal on Software Testing Verification and Reliability*, 2010.

[43] Romain Delamare, Benoit Baudry, Sudipto Ghosh, and Yves Le Traon. A test-driven approach to developing pointcut descriptors in aspectj. In *ICST (International Conference on Software Testing Verification and Validation)*, pages 376–385, Denver, CO, USA, 2009.

[44] Romain Delamare, Benoit Baudry, and Yves Le Traon. Regression test selection when evolving software with aspects. In *LATE workshop in conjunction with AOSD'08*, Brussels, Belgium, 2008.

[45] Romain Delamare, Benoit Baudry, and Yves Le Traon. A tool for the mutation analysis of aspectj pointcut descriptors. In *Mutation'09 workshop in conjunction with ICST'09*, Denver, CO, USA, 2009.

[46] Romain Delamare, Freddy Muñoz, Benoit Baudry, and Yves Le Traon. Vidock: a tool for impact analysis of aspect weaving on test cases. In *International Conference on Testing Software and Systems*, Natal, Brazil, November 2010. IFIP.

[47] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection : Help for the practicing programmer. *IEEE Computer*, 11(4):34 – 41, 1978.

[48] Philippe Dhaussy, Pierre Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, and Benoit Baudry. Evaluating context descriptions and property definition patterns for software formal validation. In *MODELS'09*, pages 438–452, Denver, CO, USA, 2009.

[49] D. F. D'Souza and A.C Wills. *Object, Components and Frameworks with UML, The Catalysis Approach*. Object Technology Series. Addison-Wesley, 1998.

[50] Keith Duddy, Anna Gerber, Michael Lawley, Kerry Raymond, and Jim Steel. Model transformation: A declarative, reusable patterns approach. In *EDOC'03 (Entreprise Distributed Object Computing Conference)*, pages 174 – 185, Brisbane, Australia, 2003.

[51] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[52] Eclipse. Ajdt: Aspectj development tools, 2010.

[53] Eclipse. Graphical modeling framework, 2010.

[54] Andy Evans, Robert B. France, Kevin Lano, and Bernhard Rumpe. The uml as a formal modeling notation. In *UML*, pages 336–348, 1998.

[55] Fabiano Cutigi Ferrari, José Carlos Maldonado, and Awais Rashid. Mutation testing for aspect-oriented programs. In *ICST'08*, pages 52 – 61, Lillehamer, Norway, 2008.

[56] Paul Karl Feyerabend. *Against Method: Outline of an Anarchistic Theory of Knowledge*. Verso, 1975.

[57] Franck Fleurey. *Langage et méthode pour une ingénierie des modèles fiable.* Phd thesis, 2006.

[58] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A generic approach for automatic model composition. In *Aspect Oriented Modeling (AOM) Workshop associated to MoDELS'07*, pages 7–15, Nashville, TN, USA, 2007.

[59] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Towards dependable model transformations: Qualifying input test data. *Software and Systems Modeling*, 8(2):185–203, 2009.

[60] Franck Fleurey, Yves Le Traon, and Benoit Baudry. From testing to diagnosis: An automated approach. In *ASE'04 (Automated Software Engineering)*, pages 306–309, Linz, Austria, 2004.

[61] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in model-driven engineering: Testing model transformations. In *MoDeVa'04 (Model Design and Validation Workshop associated to ISSRE'04)*, Rennes, France, 2004.

[62] Eclipse Foundation. Emf compare, 2007.

[63] Robert France and J.M. Bieman. Multi-view software evolution: a uml-based framework for evolving object-oriented software. In *ICSM'01 (Int. Conference on Software Maintenance)*, pages 386 – 95, Florence, Italy, 2001.

[64] Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto Ghosh. Providing support for model composition in metamodels. In *EDOC'07 (Entreprise Distributed Object Computing Conference)*, pages 253–266, Annapolis, MD, USA, 2007.

[65] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In Lionel Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE - CS Press, 2007.

[66] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[67] Carlo Ghezzi. Self managing situated computing, 2008.

[68] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering, 2nd edition*. 2002.

[69] Heather J. Goldsby and Betty H.C. Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *MODELS'08*, Toulouse, France, 2008.

[70] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52(2-3):173 – 179, 2000.

[71] Reiko Heckel and Marc Lohmann. Towards model-driven testing. *Electronic Notes in Theoretical Computer Science*, 82(6), 2003.

[72] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. Springer, 2006.

[73] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[74] Cédric Jeanneret, Martin Glinz, and Benoit Baudry. Estimating footprints of model operations. In *International Conference on Software Engineering (ICSE'11)*, Honolulu, USA, May 2011. IEEE.

[75] Jean-Marc Jézéquel. Model driven design and aspect weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2):209–218, may 2008.

[76] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE'02 (Int. Conference in Software Engineering)*, pages 467 – 477, Orlando, FL, USA, 2002.

[77] Ajay Kattepur, Sagar Sen, Benoit Baudry, Albert Benveniste, and Claude Jard. Variability modeling and qos analysis of web services orchestrations. In *International Conference on Web Services*, Miami, FL, USA, July 2010. IEEE.

[78] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 (European Conference for Object-Oriented Programming)*, 1997.

[79] Sun-Woo Kim, J.A. Clark, and J.A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Software Testing, Verification and Reliability*, 11(4):207 – 225, 2001.

[80] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI '05*, pages 1557–1560, Portland, OR, USA, 2005.

[81] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *GAMMA workshop at ICSE'06*, pages 13 – 20, Shanghai, China, 2006.

[82] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, September 2010.

[83] Jochen M. Küster and Mohamed Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *MoDeVa'06 (Model Design and Validation Workshop associated to MoDELS'06)*, Genova, Italy, 2006.

[84] Thomas Lackner. Automate your testing using bacteriological algorithms. `http://www.straatinvestments.com/blog/automate-your-testing-using-bacteriological-algorithms-3820.html`, December 2007.

[85] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In *UML'00 (Unified Modeling Language)*, volume 1939 of *Lecture Notes in Computer Science*, pages 482 – 496. Springer-Verlag, 2000.

[86] Yves Le Traon, Benoit Baudry, and Jean-Marc Jézéquel. Design by contract to improve software vigilance. *IEEE Transactions on Software Engineering*, 32(8):571–586, 2006.

[87] Yves Le Traon, Farid Ouabdessalam, Chantal Robach, and Benoit Baudry. From diagnosis to diagnosability: Axiomatization, measurement and application. *Journal of Systems and Software*, 65(1):31 – 50, 2003.

[88] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44 – 51, 2001.

[89] Y. Lin, J. Gray, and Frédéric Jouault. Dsmdiff: A differentiation tool for domain-specific models. *European Journal of Information Systems, Special Issue on Model-Driven Systems Development*, 2007.

[90] Philip K. McKinley, Betty H. C. Cheng, Charles Ofria, David B. Knoester, Benjamin E. Beckmann, and Heather Goldsby. Harnessing digital evolution. *IEEE Computer*, 41(1):54–63, 2008.

[91] MetaCase. Metaedit+ domain-specific modeling environment, 2010.

[92] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40 – 51, 1992.

[93] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1992.

[94] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin C. Rinard. Quality of service profiling. In *ICSE'10*, pages 25 – 34, Cape Town, South Africa, 2010.

[95] Ivan Moore. Jester - a junit test tester. In *XP 2001*, pages 84 – 87, Villasimius, Sardinia, 2001.

[96] Jean-Marie Mottu. *Oracles et qualification du test de transformations de modèles*. PhD thesis, Université de Rennes 1, November 2008.

[97] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation analysis testing for model transformations. In *ECMDA'06 (European Conference on Model Driven Architecture)*, pages 376–390, Bilbao, Spain, 2006.

[98] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Reusable mda components: A testing-for-trust approach. In *MoDELS'06*, pages 589–603, Genova, Italy, 2006.

[99] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Model transformation testing: oracle issue. In *MODEVVA workshop in association with ICST'08*, Lillehamer, Norway, 2008.

[100] MSDN. C# introduction and overview, 2002.

[101] MSDN. .net homepage, 2002.

[102] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS'05*, pages 264 – 278, Montego Bay, Jamaica, 2005. LNCS.

[103] Pierre-Alain Muller, Fédéric Fondement, Benoit Baudry, and Benoit Combemale. Modeling modeling modeling. *SOSYM*, 2010.

[104] Freddy Muñoz. *Validation of reasoning engines and adaptation mechanisms Validation of reasoning engines and adaptation mechanisms for self -adaptive systems*. PhD thesis, Université de Rennes 1, September 2010.

[105] Freddy Munoz, Olivier Barais, and Benoit Baudry. Vigilant usage of aspects. In *ADI workshop associated to ECOOP'07*, Berlin, Germany, 2007.

[106] Freddy Munoz, Benoit Baudry, and Olivier Barais. A classification of invasive patterns in aop. Research report, INRIA, March 2008.

[107] Freddy Munoz, Benoit Baudry, and Olivier Barais. Improving maintenance in aop through an interaction specification framework. In *ICSM'08*, pages 77–86, Beijing, China, 2008.

[108] Freddy Munoz, Benoit Baudry, Romain Delamare, and Yves Le Traon. Inquiring the usage of aspect-oriented programming: an empirical study. In *ICSM'09 (Int. Conference on Software Maintenance)*, pages 137–146, Edmonton, Al, Canada, 2009.

[109] Glenford J. Myers. *The art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[110] A. J. Offutt, Ammei Lee, G. Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99 – 118, 1996.

[111] OMG. Mda, 2003.

[112] OMG. Mof qvt final adopted specification, 2005.

[113] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676 – 686, 1988.

[114] Renaud Pawlak. Spoon, 2010.

[115] Gilles Perrouin, Erwan Brottier, Benoit Baudry, and Yves Le Traon. Composing models for detecting inconsistencies: A requirements engineering perspective. In *REFSQ'09*, pages 89–103, 2009.

[116] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *ICST'10*, Paris, France, 2010.

[117] Eric R. Pianka. *Evolutionnary Ecology*. Addison-Wesley, 1999.

[118] Karl Raimund Popper. *Conjectures and refutations: the growth of scientific knowledge*. Harper & Row, 1968.

[119] Rodrigo Ramos, Olivier Barais, and Jean-Marc Jézéquel. Matching model-snippets. In *MoDELS'07*, Nashville, TN, USA, 2007.

[120] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367 – 375, 1985.

[121] Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Probabilistic qos and soft contracts for transaction-based web services orchestrations. *IEEE T. Services Computing*, 1(4):187–200, 2008.

[122] Jeff Rothenberg. The nature of modeling. In Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen, editors, *AI, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989.

[123] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25 – 31, 2006.

[124] Sagar Sen. Cartier, 2009.

[125] Sagar Sen. *Découverte automatique de modèles effectifs*. PhD thesis, Université Rennes 1, 06 2010.

[126] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On combining multi-formalism knowledge to select models for model transformation testing. In *ICST'08 (International Conference on Software Testing Verification and Validation)*, pages 328–337, Lillehamer, Norway, 2008.

[127] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In *International Conference on Model Transformation*, pages 148–164, Zurich, Switzerland, 2009.

[128] Sagar Sen, Benoit Baudry, and Doina Precup. Partial model completion in model driven engineering using constraint logic programming. In *INAP'07 (International Conference on Applications of Declarative Programming and Knowledge Management)*, Würzburg, Germany, 2007.

[129] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Towards domain-specific model editors with automatic model completion. *Simulation*, 86(2):109 – 126, 2010.

[130] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jezequel. Meta-model pruning. In *MODELS'09*, pages 32–46, Denver, CO, USA, 2009.

[131] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, and Jean-Marc Jezequel. Reusable model transformations. *SoSym*, 2010.

[132] Jacques Simonin. *Conception de l'architecture d'un système dirigée par un modèle d'urbanisme fonctionnel*. PhD thesis, Université de Rennes 1, January 2009.

[133] Jim Steel and Jean-Marc Jézéquel. On model typing. *Software and Systems Modeling*, 6(4):401–413, 2007.

[134] Jim Steel and Jean-Marc Jézéquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, December 2007.

[135] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of OOPSLA'06*, pages 481 – 497, Portland, OR, USA, 2006.

[136] Maximilian Störzer and Jürgen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM'05*, pages 653– 656, Budapest, Hungary, 2005.

[137] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design pattern application in uml. In *ECOOP'00 (European Conference for Object-Oriented Programming)*, volume 1850 of *Lecture Notes in Computer Science*, pages 44 – 62, 2000.

[138] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computer*, 1(2):146–160, 1972.

[139] Jonathan Timmis, Andrew Hone, Thomas Stibor, and Edward Clark. Theoretical advances in artificial immune systems. *Theor. Comput. Sci.*, 403(1):11–32, 2008.

[140] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS'07*, Braga, Portugal, 2007.

[141] Tom Tourwe, Johan Brichau, and Kris Gybels. On the existence of the aosd-evolution paradox. In *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, 2003.

[142] Mark Utting and Bruno Legeard. *Practical Model-Based Testing*. Morgan Kaufmann, 2007.

[143] Terry J. van der Werff. 10 emerging technologies that will change the world. *Technology Review*, 2001.

[144] Jeffrey M. Voas and K. Miller. The revealing power of a test case. *Software Testing, Verification and Reliability*, 2(1):25 – 42, 1992.

[145] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering, ICSE*, pages 364–374, Vancouver, Canada, May 2009. IEEE.

[146] Martin Wirsing, Matthias Hölzl, and Axel Rauschmayer. Road-mapping research in software-intensive systems and new computing paradigms. Technical report, Coordination Action InterLink, March 2009.

[147] Rati Wongsathan, Isaravuth Seedadan, and Sutichart Pattarangoon;. Cluster-based routing using bacteriologic algorithm in wireless sensor network. In *Proceedings of IEEE ICCET'2010*, pages 5 – 9, Chengdu, China, April 2010.

[148] Guoqing Xu and Atanas Rountev. Regression test selection for aspectj software. In *ICSE'07 (Int. Conference in Software Engineering)*, Minneapolis, MN, USA, 2007.

[149] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[150] Sai Zhang, Zhongxian Gu, Yu Lin, and Jianjun Zhao. Change impact analysis tool for aspect-oriented programs. In *ICSM'08*, pages 87 – 96, Beijing, China, 2008.

Abstract

This habilitation introduces the question-learn-test-feedback pattern that resulted from a series of investigations in the domains of software testing and modular software construction. The contributions to these fields are of various natures, but are all fundamentally related through two major assumptions: software construction paradigms have to constantly evolve in order to deal with the increasingly complex requirements that software-intensive systems have to meet; a tester's perspective can support this evolution through the development of effective testing techniques and new empirical knowledge about these paradigms.

Abstraction, modularity and separation of concerns have been advocated as key factors for rigorous software engineering for a long time. These principles have been incarnated by various software construction paradigms such as object-oriented programming and design and model-driven development. These paradigms evolve in order to deal with the increasing number of heterogeneous requirements, the large number of variations and the need for adaptation that software-intensive systems have to integrate. The work presented here is about the integration of effective testing techniques in these paradigms and how this led us towards a more precise understanding of these paradigms.

A major discovery in our work is that we could follow a systematic pattern when investigating these paradigms to integrate error detection capabilities. First, we have to question these paradigms about the new assumptions they introduce on software systems. When answering these questions we can can perform the following actions: learn through rigorous evaluation of hypotheses about these paradigms; test software systems developed in these new paradigms; provide feedback to the paradigms in the form of new construction techniques that improve testability. We capture these four facets for the investigation of software construction paradigms in the question-learn-test-feedback pattern (QLTF).

This habilitation reports on investigations in three software construction techniques: object-oriented programming and design, aspect-oriented programming, model transformations. Each investigation is synthesized around the question-learn-test-feedback pattern.